SECOND EDITION

COMPUTER SCIENCE

HANDBOOK

EDITOR-IN-CHIEF

ALLEN B. TUCKER

CHAPMAN & HALL/CRC

Published in Cooperation with ACM, The Association for Computing Machinery

Library of Congress Cataloging-in-Publication Data

Computer science handbook / editor-in-chief, Allen B. Tucker—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 1-58488-360-X (alk. paper)
1.Computer science-Handbooks, manuals, etc. 2. Engineering—Hanbooks, manuals, etc.
I. Tucker, Allen B.

QA76.C54755 2004
004—dc22 2003068758

Direct all inquiries to CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431.

International Standard Book Number 1-58488-360-X
Library of Congress Card Number 2003068758
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

Preface to the Second Edition

Purpose

The purpose of *The Computer Science Handbook* is to provide a single comprehensive reference for computer scientists, software engineers, and IT professionals who wish to broaden or deepen their understanding in a particular subfield of computer science. Our goal is to provide the most current information in each of the following eleven subfields in a form that is accessible to students, faculty, and professionals in computer science:

> algorithms, architecture, computational science, graphics, human-computer interaction, information management, intelligent systems, net-centric computing, operating systems, programming languages, and software engineering

Each of the eleven sections of the *Handbook* is dedicated to one of these subfields. In addition, the appendices provide useful information about professional organizations in computer science, standards, and languages. Different points of access to this rich collection of theory and practice are provided through the table of contents, two introductory chapters, a comprehensive subject index, and additional indexes.

A more complete overview of this *Handbook* can be found in Chapter 1, which summarizes the contents of each of the eleven sections. This chapter also provides a history of the evolution of computer science during the last 50 years, as well as its current status, and future prospects.

New Features

Since the first edition of the *Handbook* was published in 1997, enormous changes have taken place in the discipline of computer science. The goals of the second edition of the *Handbook* are to incorporate these changes by:

1. Broadening its reach across all 11 subject areas of the discipline, as they are defined in *Computing Curricula 2001* (the new standard taxonomy)
2. Including a heavier proportion of applied computing subject matter
3. Bringing up to date all the topical discussions that appeared in the first edition

This new edition was developed by the editor-in-chief and three editorial advisors, whereas the first edition was developed by the editor and ten advisors. Each edition represents the work of over 150 contributing authors who are recognized as experts in their various subfields of computer science.

Readers who are familiar with the first edition will notice the addition of many new chapters, reflecting the rapid emergence of new areas of research and applications since the first edition was published. Especially exciting are the addition of new chapters in the areas of computational science, information

management, intelligent systems, net-centric computing, and software engineering. These chapters explore topics like cryptography, computational chemistry, computational astrophysics, human-centered software development, cognitive modeling, transaction processing, data compression, scripting languages, multimedia databases, event-driven programming, and software architecture.

Acknowledgments

A work of this magnitude cannot be completed without the efforts of many individuals. During the 2-year process that led to the first edition, I had the pleasure of knowing and working with ten very distinguished, talented, and dedicated editorial advisors:

> Harold Abelson (MIT), Mikhail Atallah (Purdue), Keith Barker (Uconn), Kim Bruce (Williams), John Carroll (VPI), Steve Demurjian (Uconn), Donald House (Texas A&M), Raghu Ramakrishnan (Wisconsin), Eugene Spafford (Purdue), Joe Thompson (Mississippi State), and Peter Wegner (Brown).

For this edition, a new team of trusted and talented editorial advisors helped to reshape and revitalize the *Handbook* in valuable ways:

> Robert Cupper (Allegheny), Fadi Deek (NJIT), Robert Noonan (William and Mary)

All of these persons provided valuable insights into the substantial design, authoring, reviewing, and production processes throughout the first eight years of this *Handbook*'s life, and I appreciate their work very much.

Of course, it is the chapter authors who have shared in these pages their enormous expertise across the wide range of subjects in computer science. Their hard work in preparing and updating their chapters is evident in the very high quality of the final product. The names of all chapter authors and their current professional affiliations are listed in the contributor list.

I want also to thank Bowdoin College for providing institutional support for this work. Personal thanks go especially to Craig McEwen, Sue Theberge, Matthew Jacobson-Carroll, Alice Morrow, and Aaron Olmstead at Bowdoin, for their various kinds of support as this project has evolved over the last eight years. Bob Stern, Helena Redshaw, Joette Lynch, and Robert Sims at CRC Press also deserve thanks for their vision, perseverance and support throughout this period.

Finally, the greatest thanks is always reserved for my wife Meg – my best friend and my love – for her eternal influence on my life and work.

Allen B. Tucker
Brunswick, Maine

Editor-in-Chief

Allen B. Tucker is the Anne T. and Robert M. Bass Professor of Natural Sciences in the Department of Computer Science at Bowdoin College, where he has taught since 1988. Prior to that, he held similar positions at Colgate and Georgetown Universities. Overall, he has served eighteen years as a department chair and two years as an associate dean. At Colgate, he held the John D. and Catherine T. MacArthur Chair in Computer Science.

Professor Tucker earned a B.A. in mathematics from Wesleyan University in 1963 and an M.S. and Ph.D. in computer science from Northwestern University in 1970. He is the author or coauthor of several books and articles in the areas of programming languages, natural language processing, and computer science education. He has given many talks, panel discussions, and workshop presentations in these areas, and has served as a reviewer for various journals, NSF programs, and curriculum projects. He has also served as a consultant to colleges, universities, and other institutions in the areas of computer science curriculum, software design, programming languages, and natural language processing applications.

A Fellow of the ACM, Professor Tucker co-authored the 1986 Liberal Arts Model Curriculum in Computer Science and co-chaired the ACM/IEEE-CS Joint Curriculum Task Force that developed Computing Curricula 1991. For these and other related efforts, he received the ACM's 1991 Outstanding Contribution Award, shared the IEEE's 1991 Meritorious Service Award, and received the ACM SIGCSE's 2001 Award for Outstanding Contributions to Computer Science Education. In Spring 2001, he was a Fulbright Lecturer at the Ternopil Academy of National Economy (TANE) in Ukraine. Professor Tucker has been a member of the ACM, the NSF CISE Advisory Committee, the IEEE Computer Society, Computer Professionals for Social Responsibility, and the Liberal Arts Computer Science (LACS) Consortium.

Contributors

Eric W. Allender
Rutgers University

James L. Alty
Loughborough University

Thomas E. Anderson
University of Washington

M. Pauline Baker
National Center for Supercomputing Applications

Steven Bellovin
AT&T Research Labs

Andrew P. Bernat
Computer Research Association

Brian N. Bershad
University of Washington

Christopher M. Bishop
Microsoft Research

Guy E. Blelloch
Carnegie Mellon University

Philippe Bonnet
University of Copenhagen

Jonathan P. Bowen
London South Bank University

Kim Bruce
Williams College

Steve Bryson
NASA Ames Research Center

Douglas C. Burger
University of Wisconsin at Madison

Colleen Bushell
National Center for Supercomputing Applications

Derek Buzasi
U.S. Air Force Academy

William L. Bynum
College of William and Mary

Bryan M. Cantrill
Sun Microsystems, Inc.

Luca Cardelli
Microsoft Research

David A. Caughy
Cornell University

Vijay Chandru
Indian Institute of Science

Steve J. Chapin
Syracuse University

Eric Chown
Bowdoin College

Jacques Cohen
Brandeis University

J.L. Cox
Brooklyn College, CUNY

Alan B. Craig
National Center for Supercomputing Applications

Maxime Crochemore
University of Marne-la-Vallée and King's College London

Robert D. Cupper
Allegheny College

Thomas Dean
Brown Univeristy

Fadi P. Deek
New Jersey Institute of Technology

Gerald DeJong
University of Illinois at Urbana-Champaign

Steven A. Demurjian Sr.
University of Connecticut

Peter J. Denning
Naval Postgraduate School

Angel Diaz
IBM Research

T.W. Doeppner Jr.
Brown University

Henry Donato
College of Charleston

Chitra Dorai
IBM T.J. Watson Research Center

Wolfgang Dzida
Pro Context GmbH

David S. Ebert
Purdue University

Raimund Ege
Florida International University

Osama Eljabiri
New Jersey Institute of Technology

David Ferbrache
U.K. Ministry of Defence

Raphael Finkel
University of Kentucky

John M. Fitzgerald
Adept Technology

Michael J. Flynn
Stanford University

Kenneth D. Forbus
Northwestern University

Stephanie Forrest
University of New Mexico

Michael J. Franklin
University of California at Berkeley

John D. Gannon
University of Maryland

Carlo Ghezzi
Politecnico di Milano

Benjamin Goldberg
New York University

James R. Goodman
University of Wisconsin at Madison

Jonathan Grudin
Microsoft Research

Gamil A. Guirgis
College of Charleston

Jon Hakkila
College of Charleston

Sandra Harper
College of Charleston

Frederick J. Heldrich
College of Charleston

Katherine G. Herbert
New Jersey Institute of Technology

Michael G. Hinchey
NASA Goddard Space Flight Center

Ken Hinckley
Microsoft Research

Donald H. House
Texas A&M University

Windsor W. Hsu
IBM Research

Daniel Huttenlocher
Cornell University

Yannis E. Ioannidis
University of Wisconsin

Robert J.K. Jacob
Tufts University

Sushil Jajodia
George Mason University

Mehdi Jazayeri
Technical University of Vienna

Tao Jiang
University of California

Michael J. Jipping
Hope College

Deborah G. Johnson
University of Virginia

Michael I. Jordan
University of California at Berkeley

David R. Kaeli
Northeastern University

Erich Kaltófen
North Carolina State University

Subbarao Kambhampati
Arizona State University

Lakshmi Kantha
University of Colorado

Gregory M. Kapfhammer
Allegheny College

Jonathan Katz
University of Maryland

Arie Kaufman
State University of New York at Stony Brook

Samir Khuller
University of Maryland

David Kieras
University of Michigan

David T. Kingsbury
Gordon and Betty Moore Foundation

Danny Kopec
Brooklyn College, CUNY

Henry F. Korth
Lehigh University

Kristin D. Krantzman
College of Charleston

Edward D. Lazowska
University of Washington

Thierry Lecroq
University of Rouen

D.T. Lee
Northwestern University

Miriam Leeser
Northeastern University

Henry M. Levy
University of Washington

Frank L. Lewis
University of Texas at Arlington

Ming Li
University of Waterloo

Ying Li
IBM T.J. Watson
Research Center

Jianghui Liu
New Jersey Institute
of Technology

Kai Liu
Alcatel Telecom

Kenneth C. Louden
San Jose State University

Michael C. Loui
University of Illinois at
Urbana-Champaign

James J. Lu
Emory University

Abby Mackness
Booz Allen Hamilton

Steve Maddock
University of Sheffield

Bruce M. Maggs
Carnegie Mellon University

Dino Mandrioli
Politecnico di Milano

M. Lynne Markus
Bentley College

Tony A. Marsland
University of Alberta

Edward J. McCluskey
Stanford University

James A. McHugh
New Jersey Institute
of Technology

Marshall Kirk McKusick
Consultant

Clyde R. Metz
College of Charleston

Keith W. Miller
University of Illinois

Subhasish Mitra
Stanford University

Stuart Mort
U.K. Defence and Evaluation
Research Agency

Rajeev Motwani
Stanford University

Klaus Mueller
State University of New York
at Stony Brook

Sape J. Mullender
Lucent Technologies

Brad A. Myers
Carnegie Mellon University

Peter G. Neumann
SRI International

Jakob Nielsen
Nielsen Norman Group

Robert E. Noonan
College of William and Mary

Ahmed K. Noor
Old Dominion University

Vincent Oria
New Jersey Institute
of Technology

Jason S. Overby
College of Charleston

M. Tamer Özsu
University of Waterloo

Victor Y. Pan
Lehman College, CUNY

Judea Pearl
University of California
at Los Angeles

Jih-Kwon Peir
University of Florida

Radia Perlman
Sun Microsystems Laboratories

Patricia Pia
University of Connecticut

Steve Piacsek
Naval Research Laboratory

Roger S. Pressman
R.S. Pressman & Associates,
Inc.

J. Ross Quinlan
University of New South Wales

Balaji Raghavachari
University of Texas at Dallas

Prabhakar Raghavan
Verity, Inc.

Z. Rahman
College of William and Mary

M.R. Rao
Indian Institute of Management

Bala Ravikumar
University of Rhode Island

Kenneth W. Regan
State University of New York at Buffalo

Edward M. Reingold
Illinois Institute of Technology

Alyn P. Rockwood
Colorado School of Mines

Robert S. Roos
Allegheny College

Erik Rosenthal
University of New Haven

Kevin W. Rudd
Intel, Inc.

Betty Salzberg
Northeastern University

Pierangela Samarati
Universitá degli Studi di Milano

Ravi S. Sandhu
George Mason University

David A. Schmidt
Kansas State University

Stephen B. Seidman
New Jersey Institute of Technology

Stephanie Seneff
Massachusetts Institute of Technology

J.S. Shang
Air Force Research

Dennis Shasha
Courant Institute
New York University

William R. Sherman
National Center for Supercomputing Applications

Avi Silberschatz
Yale University

Gurindar S. Sohi
University of Wisconsin at Madison

Ian Sommerville
Lancaster University

Bharat K. Soni
Mississippi State University

William Stallings
Consultant and Writer

John A. Stankovic
University of Virginia

S. Sudarshan
IIT Bombay

Earl E. Swartzlander Jr.
University of Texas at Austin

Roberto Tamassia
Brown University

Patricia J. Teller
University of Texas at ElPaso

Robert J. Thacker
McMaster University

Nadia Magnenat Thalmann
University of Geneva

Daniel Thalmann
Swiss Federal Institute of Technology (EPFL)

Alexander Thomasian
New Jersey Institute of Technology

Allen B. Tucker
Bowdoin College

Jennifer Tucker
Booz Allen Hamilton

Patrick Valduriez
INRIA and IRIN

Jason T.L. Wang
New Jersey Institute of Technology

Colin Ware
University of New Hampshire

Alan Watt
University of Sheffield

Nigel P. Weatherill
University of Wales Swansea

Peter Wegner
Brown University

Jon B. Weissman
University of Minnesota-Twin Cities

Craig E. Wills
Worcester Polytechnic Institute

George Wolberg
City College of New York

Donghui Zhang
Northeastern University

Victor Zue
Massachusetts Institute of Technology

Contents

Section III: Computational Science

Section IV: Graphics and Visual Computing

Section VIII: Net-Centric Computing

Section IX: Operating Systems

Section X: Programming Languages

Section XI: Software Engineering

1

Computer Science: The Discipline and its Impact

Allen B. Tucker
Bowdoin College

Peter Wegner
Brown University

1.1 Introduction

The field of computer science has undergone a dramatic evolution in its short 70-year life. As the field has matured, new areas of research and applications have emerged and joined with classical discoveries in a continuous cycle of revitalization and growth.

In the 1930s, fundamental mathematical principles of computing were developed by Turing and Church. Early computers implemented by von Neumann, Wilkes, Eckert, Atanasoff, and others in the 1940s led to the birth of scientific and commercial computing in the 1950s, and to mathematical programming languages like Fortran, commercial languages like COBOL, and artificial-intelligence languages like LISP. In the 1960s the rapid development and consolidation of the subjects of algorithms, data structures, databases, and operating systems formed the core of what we now call traditional computer science; the 1970s saw the emergence of software engineering, structured programming, and object-oriented programming. The emergence of personal computing and networks in the 1980s set the stage for dramatic advances in computer graphics, software technology, and parallelism. The 1990s saw the worldwide emergence of the Internet, both as a medium for academic and scientific exchange and as a vehicle for international commerce and communication.

This Handbook aims to characterize computer science in the new millenium, incorporating the explosive growth of the Internet and the increasing importance of subject areas like human–computer interaction, massively parallel scientific computation, ubiquitous information technology, and other subfields that

would not have appeared in such an encyclopedia even ten years ago. We begin with the following short definition, a variant of the one offered in [Gibbs 1986], which we believe captures the essential nature of "computer science" as we know it today.

> *Computer science* is the study of computational processes and information structures, including their hardware realizations, their linguistic models, and their applications.

The Handbook is organized into eleven sections which correspond to the eleven major subject areas that characterize computer science [ACM/IEEE 2001], and thus provide a useful modern taxonomy for the discipline. The next section presents a brief history of the computing industry and the parallel development of the computer science curriculum. Section 1.3 frames the practice of computer science in terms of four major conceptual paradigms: theory, abstraction, design, and the social context. Section 1.4 identifies the "grand challenges" of computer science research and the subsequent emergence of information technology and cyber-infrastructure that may provide a foundation for addressing these challenges during the next decade and beyond. Section 1.5 summarizes the subject matter in each of the Handbook's eleven sections in some detail.

This Handbook is designed as a professional reference for researchers and practitioners in computer science. Readers interested in exploring specific subject topics may prefer to move directly to the appropriate section of the Handbook — the chapters are organized with minimal interdependence, so that they can be read in any order. To facilitate rapid inquiry, the Handbook contains a Table of Contents and three indexes (Subject, Who's Who, and Key Algorithms and Formulas), providing access to specific topics at various levels of detail.

1.2 Growth of the Discipline and the Profession

The computer industry has experienced tremendous growth and change over the past several decades. The transition that began in the 1980s, from centralized mainframes to a decentralized networked microcomputer–server technology, was accompanied by the rise and decline of major corporations. The old monopolistic, vertically integrated industry epitomized by IBM's comprehensive client services gave way to a highly competitive industry in which the major players changed almost overnight. In 1992 alone, emergent companies like Dell and Microsoft had spectacular profit gains of 77% and 53%. In contrast, traditional companies like IBM and Digital suffered combined record losses of $7.1 billion in the same year [Economist 1993] (although IBM has since recovered significantly). As the 1990s came to an end, this euphoria was replaced by concerns about new monopolistic behaviors, expressed in the form of a massive antitrust lawsuit by the federal government against Microsoft. The rapid decline of the "dot.com" industry at the end of the decade brought what many believe a long-overdue rationality to the technology sector of the economy. However, the exponential decrease in computer cost and increase in power by a factor of two every 18 months, known as Moore's law, shows no signs of abating in the near future, although underlying physical limits will eventually be reached.

Overall, the rapid 18% annual growth rate that the computer industry had enjoyed in earlier decades gave way in the early 1990s to a 6% growth rate, caused in part by a saturation of the personal computer market. Another reason for this slowing of growth is that the performance of computers (speed, storage capacity) has improved at a rate of 30% per year in relation to their cost. Today, it is not unusual for a laptop or hand-held computer to run at hundreds of times the speed and capacity of a typical computer of the early 1990s, and at a fraction of its cost. However, it is not clear whether this slowdown represents a temporary plateau or whether a new round of fundamental technical innovations in areas such as parallel architectures, nanotechnology, or human–computer interaction might generate new spectacular rates of growth in the future.

1.2.1 Curriculum Development

The computer industry's evolution has always been affected by advances in both the theory and the practice of computer science. Changes in theory and practice are simultaneously intertwined with the evolution of the field's undergraduate and graduate curricula, which have served to define the intellectual and methodological framework for the discipline of computer science itself.

The first coherent and widely cited curriculum for computer science was developed in 1968 by the ACM Curriculum Committee on Computer Science [ACM 1968] in response to widespread demand for systematic undergraduate and graduate programs [Rosser 1966]. "Curriculum 68" defined computer science as comprising three main areas: information structures and processes, information processing systems, and methodologies. Curriculum 68 defined computer science as a discipline and provided concrete recommendations and guidance to colleges and universities in developing undergraduate, master's, and doctorate programs to meet the widespread demand for computer scientists in research, education, and industry. Curriculum 68 stood as a robust and exemplary model for degree programs at all levels for the next decade.

In 1978, a new ACM Curriculum Committee on Computer Science developed a revised and updated undergraduate curriculum [ACM 1978]. The "Curriculum 78" report responded to the rapid evolution of the discipline and the practice of computing, and to a demand for a more detailed elaboration of the computer science (as distinguished from the mathematical) elements of the courses that would comprise the core curriculum.

During the next few years, the IEEE Computer Society developed a model curriculum for engineering-oriented undergraduate programs [IEEE-CS 1976], updated and published it in 1983 as a "Model Program in Computer Science and Engineering" [IEEE-CS 1983], and later used it as a foundation for developing a new set of accreditation criteria for undergraduate programs. A simultaneous effort by a different group resulted in the design of a model curriculum for computer science in liberal arts colleges [Gibbs 1986]. This model emphasized science and theory over design and applications, and it was widely adopted by colleges of liberal arts and sciences in the late 1980s and the 1990s.

In 1988, the ACM Task Force on the Core of Computer Science and the IEEE Computer Society [ACM 1988] cooperated in developing a fundamental redefinition of the discipline. Called "Computing as a Discipline," this report aimed to provide a contemporary foundation for undergraduate curriculum design by responding to the changes in computing research, development, and industrial applications in the previous decade. This report also acknowledged some fundamental methodological changes in the field. The notion that "computer science = programming" had become wholly inadequate to encompass the richness of the field. Instead, three different paradigms—called *theory, abstraction,* and *design*—were used to characterize how various groups of computer scientists did their work. These three points of view — those of the theoretical mathematician or scientist (theory), the experimental or applied scientist (abstraction, or modeling), and the engineer (design) — were identified as essential components of research and development across all nine subject areas into which the field was then divided.

"Computing as a Discipline" led to the formation of a joint ACM/IEEE-CS Curriculum Task Force, which developed a more comprehensive model for undergraduate curricula called "Computing Curricula 91" [ACM/IEEE 1991]. Acknowledging that computer science programs had become widely supported in colleges of engineering, arts and sciences, and liberal arts, Curricula 91 proposed a core body of knowledge that undergraduate majors in all of these programs should cover. This core contained sufficient theory, abstraction, and design content that students would become familiar with the three complementary ways of "doing" computer science. It also ensured that students would gain a broad exposure to the nine major subject areas of the discipline, including their social context. A significant laboratory component ensured that students gained significant abstraction and design experience.

In 2001, in response to dramatic changes that had occurred in the discipline during the 1990s, a new ACM/IEEE-CS Task Force developed a revised model curriculum for computer science [ACM/IEEE 2001]. This model updated the list of major subject areas, and we use this updated list to form the organizational basis for this Handbook (see below). This model also acknowledged that the enormous

growth of the computing field had spawned four distinct but overlapping subfields — "computer science," "computer engineering," "software engineering," and "information systems." While these four subfields share significant knowledge in common, each one also underlies a distinctive academic and professional field. While the computer science dimension is directly addressed by this Handbook, the other three dimensions are addressed to the extent that their subject matter overlaps that of computer science.

1.2.2 Growth of Academic Programs

Fueling the rapid evolution of curricula in computer science during the last three decades was an enormous growth in demand, by industry and academia, for computer science professionals, researchers, and educators at all levels. In response, the number of computer science Ph.D.-granting programs in the U.S. grew from 12 in 1964 to 164 in 2001. During the period 1966 to 2001, the annual number of Bachelor's degrees awarded in the U.S. grew from 89 to 46,543; Master's degrees grew from 238 to 19,577; and Ph.D. degrees grew from 19 to 830 [ACM 1968, Bryant 2001].

Figure 1.1 shows the number of bachelor's and master's degrees awarded by U.S. colleges and universities in computer science and engineering (CS&E) from 1966 to 2001. The number of Bachelor's degrees peaked at about 42,000 in 1986, declined to about 24,500 in 1995, and then grew steadily toward its current peak during the past several years. Master's degree production in computer science has grown steadily without decline throughout this period.

The dramatic growth of BS and MS degrees in the five-year period between 1996 and 2001 parallels the growth and globalization of the economy itself. The more recent falloff in the economy, especially the collapse of the "dot.com" industry, may dampen this growth in the near future. In the long run, future increases in Bachelor's and Master's degree production will continue to be linked to expansion of the technology industry, both in the U.S and throughout the world.

Figure 1.2 shows the number of U.S. Ph.D. degrees in computer science during the same 1966 to 2001 period [Bryant 2001]. Production of Ph.D. degrees in computer science grew throughout the early 1990s, fueled by continuing demand from industry for graduate-level talent and from academia to staff growing undergraduate and graduate research programs. However, in recent years, Ph.D. production has fallen off slightly and approached a steady state. Interestingly, this last five years of non-growth at the Ph.D. level is coupled with five years of dramatic growth at the BS and MS levels. This may be partially explained by the unusually high salaries offered in a booming technology sector of the economy, which may have lured some

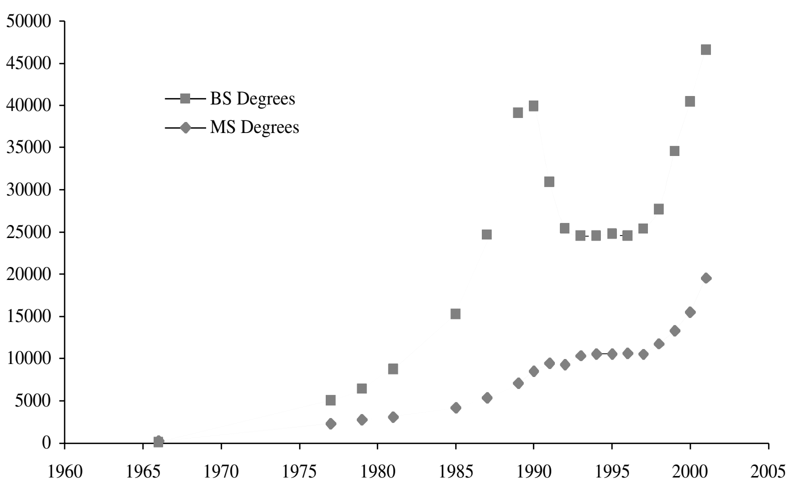


FIGURE 1.1 U.S. bachelor's and master's degrees in CS&E.

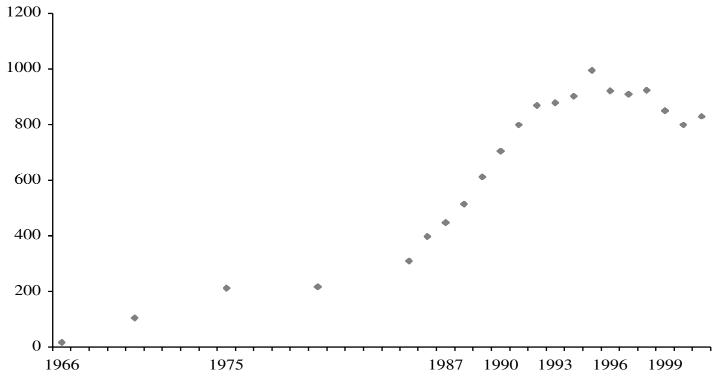


FIGURE 1.2 U.S. Ph.D. degrees in computer science.

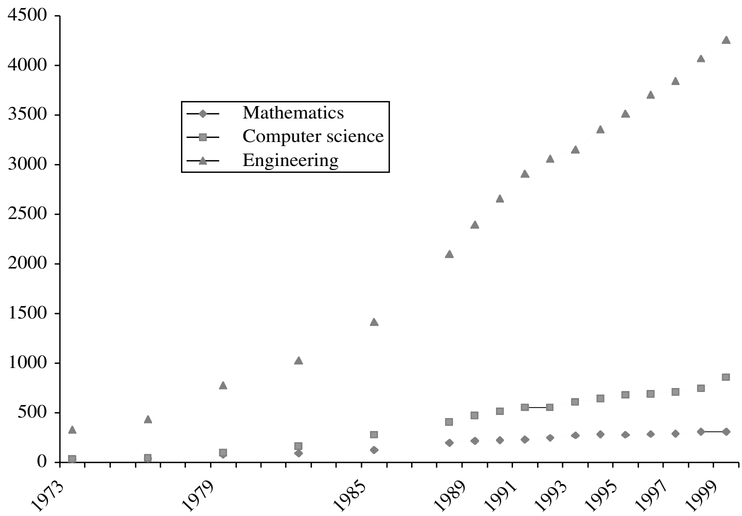


FIGURE 1.3 Academic R&D in computer science and related fields (in millions of dollars).

undergraduates away from immediate pursuit of a Ph.D. The more recent economic slowdown, especially in the technology industry, may help to normalize these trends in the future.

1.2.3 Academic R&D and Industry Growth

University and industrial research and development (R&D) investments in computer science grew rapidly in the period between 1986 and 1999. Figure 1.3 shows that academic research and development in computer science nearly tripled, from $321 million to $860 million, during this time period. This growth rate was significantly higher than that of academic R&D in the related fields of engineering and mathematics. During this same period, the overall growth of academic R&D in engineering doubled, while that in mathematics grew by about 50%. About two thirds of the total support for academic R&D comes from federal and state sources, while about 7% comes from industry and the rest comes from the academic institutions themselves [NSF 2002].

Using 1980, 1990, and 2000 U.S. Census data, Figure 1.4 shows recent growth in the number of persons with at least a bachelor's degree who were employed in nonacademic (industry and government) computer

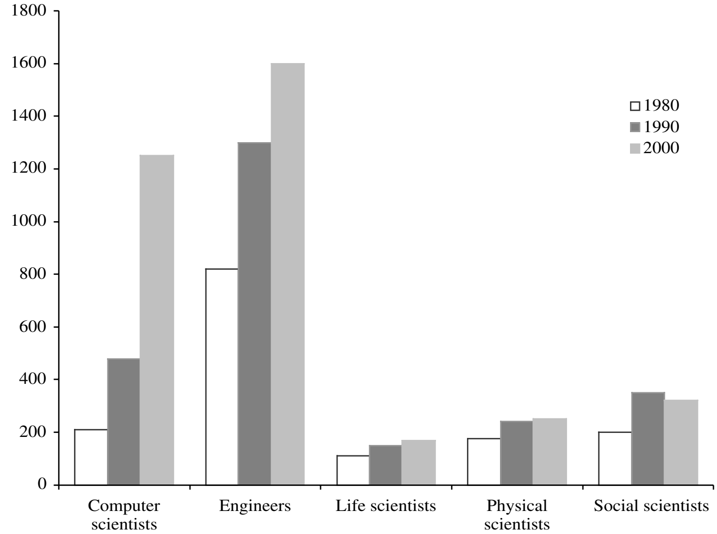


FIGURE 1.4 Nonacademic computer scientists and other professions (thousands).

science positions. Overall, the total number of computer scientists in these positions grew by 600%, from 210,000 in 1980 to 1,250,000 in 2000. Surveys conducted by the Computing Research Association (CRA) suggest that about two thirds of the domestically employed new Ph.D.s accept positions in industry or government, and the remainder accept faculty and postdoctoral research positions in colleges and universities.

CRA surveys also suggest that about one third of the total number of computer science Ph.D.s accept positions abroad [Bryant 2001]. Coupled with this trend is the fact that increasing percentages of U.S. Ph.D.s are earned by non-U.S. citizens. In 2001, about 50% of the total number of Ph.D.s were earned by this group.

Figure 1.4 also provides nonacademic employment data for other science and engineering professions, again considering only persons with bachelor's degrees or higher. Here, we see that all areas grew during this period, with computer science growing at the highest rate. In this group, only engineering had a higher total number of persons in the workforce, at 1.6 million. Overall, the total nonacademic science and engineering workforce grew from 2,136,200 in 1980 to 3,664,000 in 2000, an increase of about 70% [NSF 2001].

1.3 Perspectives in Computer Science

By its very nature, computer science is a multifaceted discipline that can be viewed from at least four different perspectives. Three of the perspectives — *theory, abstraction,* and *design* — underscore the idea that computer scientists in all subject areas can approach their work from different intellectual viewpoints and goals. A fourth perspective — *the social and professional context* — acknowledges that computer science applications directly affect the quality of people's lives, so that computer scientists must understand and confront the social issues that their work uniquely and regularly encounters.

The *theory* of computer science draws from principles of mathematics as well as from the formal methods of the physical, biological, behavioral, and social sciences. It normally includes the use of abstract ideas and methods taken from subfields of mathematics such as logic, algebra, analysis, and statistics. Theory includes the use of various proof and argumentation techniques, like induction and contradiction, to establish properties of formal systems that justify and explain underlying the basic algorithms and data structures used in computational models. Examples include the study of algorithmically unsolvable problems and the study of upper and lower bounds on the complexity of various classes of algorithmic problems. Fields like algorithms and complexity, intelligent systems, computational science, and programming languages have different theoretical models than human–computer interaction or net-centric computing; indeed, all 11 areas covered in this Handbook have underlying theories to a greater or lesser extent.

Abstraction in computer science includes the use of scientific inquiry, modeling, and experimentation to test the validity of hypotheses about computational phenomena. Computer professionals in all 11 areas of the discipline use abstraction as a fundamental tool of inquiry — many would argue that computer science is itself the science of building and examining abstract computational models of reality. Abstraction arises in computer architecture, where the Turing machine serves as an abstract model for complex real computers, and in programming languages, where simple semantic models such as lambda calculus are used as a framework for studying complex languages. Abstraction appears in the design of heuristic and approximation algorithms for problems whose optimal solutions are computationally intractable. It is surely used in graphics and visual computing, where models of three-dimensional objects are constructed mathematically; given properties of lighting, color, and surface texture; and projected in a realistic way on a two-dimensional video screen.

Design is a process that models the essential structure of complex systems as a prelude to their practical implementation. It also encompasses the use of traditional engineering methods, including the classical life-cycle model, to implement efficient and useful computational systems in hardware and software. It includes the use of tools like cost/benefit analysis of alternatives, risk analysis, and fault tolerance that ensure that computing applications are implemented effectively. Design is a central preoccupation of computer architects and software engineers who develop hardware systems and software applications. Design is an especially important activity in computational science, information management, human–computer interaction, operating systems, and net-centric computing.

The *social and professional context* includes many concerns that arise at the computer–human interface, such as liability for hardware and software errors, security and privacy of information in databases and networks (e.g., implications of the Patriot Act), intellectual property issues (e.g., patent and copyright), and equity issues (e.g., universal access to technology and to the profession). All computer scientists must consider the ethical context in which their work occurs and the special responsibilities that attend their work. Chapter 2 discusses these issues, and Appendix B presents the ACM Code of Ethics and Professional Conduct. Several other chapters address topics in which specific social and professional issues come into play. For example, security and privacy issues in databases, operating systems, and networks are discussed in Chapter 60 and Chapter 77. Risks in software are discussed in several chapters of Section XI.

1.4 Broader Horizons: From HPCC to Cyberinfrastructure

In 1989, the Federal Office of Science and Technology announced the "High Performance Computing and Communications Program," or HPCC [OST 1989]. HPCC was designed to encourage universities, research programs, and industry to develop specific capabilities to address the "grand challenges" of the future. To realize these grand challenges would require both fundamental and applied research, including the development of high-performance computing systems with speeds two to three orders of magnitude greater than those of current systems, advanced software technology and algorithms that enable scientists and mathematicians to effectively address these grand challenges, networking to support R&D for a gigabit National Research and Educational Network (NREN), and human resources that expand basic research in all areas relevant to high-performance computing.

The grand challenges themselves were identified in HPCC as those fundamental problems in science and engineering with potentially broad economic, political, or scientific impact that can be advanced by applying high-performance computing technology and that can be solved only by high-level collaboration among computer professionals, scientists, and engineers. A list of grand challenges developed by agencies such as the NSF, DoD, DoE, and NASA in 1989 included:

- Prediction of weather, climate, and global change
- Challenges in materials sciences
- Semiconductor design
- Superconductivity
- Structural biology

- Design of drugs
- Human genome
- Quantum chromodynamics
- Astronomy
- Transportation
- Vehicle dynamics and signature
- Turbulence
- Nuclear fusion
- Combustion systems
- Oil and gas recovery
- Ocean science
- Speech
- Vision
- Undersea surveillance for anti-submarine warfare

The 1992 report entitled "Computing the Future" (CTF) [CSNRCTB 1992], written by a group of leading computer professionals in response to a request by the Computer Science and Technology Board (CSTB), identified the need for computer science to broaden its research agenda and its educational horizons, in part to respond effectively to the grand challenges identified above. The view that the research agenda should be broadened caused concerns among some researchers that this funding and other incentives might overemphasize short-term at the expense of long-term goals. This Handbook reflects the broader view of the discipline in its inclusion of computational science, information management, and human–computer interaction among the major subfields of computer science.

CTF aimed to bridge the gap between suppliers of research in computer science and consumers of research such as industry, the federal government, and funding agencies such as the NSF, DARPA, and DoE. It addressed fundamental challenges to the field and suggested responses that encourage greater interaction between research and computing practice. Its overall recommendations focused on three priorities:

1. To sustain the core effort that creates the theoretical and experimental science base on which applications build
2. To broaden the field to reflect the centrality of computing in science and society
3. To improve education at both the undergraduate and graduate levels

CTF included recommendations to federal policy makers and universities regarding research and education:

- *Recommendations to federal policy makers regarding research:*
  – The High-Performance Computing and Communication (HPCC) program passed by Congress in 1989 [OST 1989] should be fully supported.
  – Application-oriented computer science and engineering research should be strongly encouraged through special funding programs.
- *Recommendations to universities regarding research:*
  – Academic research should broaden its horizons, embracing application-oriented and technology-transfer research as well as core applications.
  – Laboratory research with experimental as well as theoretical content should be supported.
- *Recommendation to federal policy makers regarding education:*
  – Basic and human resources research of HPCC and other areas should be expanded to address educational needs.

- *Recommendations to universities regarding education:*
  - Broaden graduate education to include requirements and incentives to study application areas.
  - Reach out to women and minorities to broaden the talent pool.

Although this report was motivated by the desire to provide a rationale for the HPCC program, its message that computer science must be responsive to the needs of society is much broader. The years since publication of CTF have seen a swing away from pure research toward application-oriented research that is reflected in this edition of the Handbook. However, it remains important to maintain a balance between short-term applications and long-term research in traditional subject areas.

More recently, increased attention has been paid to the emergence of information technology (IT) research as an academic subject area having significant overlap with computer science itself. This development is motivated by several factors, including mainly the emergence of electronic commerce, the shortage of trained IT professionals to fill new jobs in IT, and the continuing need for computing to expand its capability to manage the enormous worldwide growth of electronic information. Several colleges and universities have established new IT degree programs that complement their computer science programs, offering mainly BS and MS degrees in information technology. The National Science Foundation is a strong supporter of IT research, earmarking $190 million in this priority area for FY 2003. This amounts to about 35% of the entire NSF computer science and engineering research budget [NSF 2003a].

The most recent initiative, dubbed "Cyberinfrastructure" [NSF 2003b], provides a comprehensive vision for harnessing the fast-growing technological base to better meet the new challenges and complexities that are shared by a widening community of researchers, professionals, organizations, and citizens who use computers and networks every day. Here are some excerpts from the executive summary for this initiative:

> . . . a new age has dawned in scientific and engineering research, pushed by continuing progress in computing, information, and communication technology, and pulled by the expanding complexity, scope, and scale of today's challenges. The capacity of this technology has crossed thresholds that now make possible a comprehensive "cyberinfrastructure" on which to build new types of scientific and engineering knowledge environments and organizations and to pursue research in new ways and with increased efficacy.
>
> Such environments . . . are required to address national and global priorities, such as understanding global climate change, protecting our natural environment, applying genomics-proteomics to human health, maintaining national security, mastering the world of nanotechnology, and predicting and protecting against natural and human disasters, as well as to address some of our most fundamental intellectual questions such as the formation of the universe and the fundamental character of matter.
>
> This panel's overarching recommendation is that the NSF should establish and lead a large-scale, interagency, and internationally coordinated Advanced Cyberinfrastructure Program (ACP) to create, deploy, and apply cyberinfrastructure in ways that radically empower all scientific and engineering research and allied education. We estimate that sustained new NSF funding of $1 billion per year is needed to achieve critical mass and to leverage the coordinated co-investment from other federal agencies, universities, industry, and international sources necessary to empower a revolution.

It is too early to tell whether the ambitions expressed in this report will provide a new rallying call for science and technology research in the next decade. Achieving them will surely require unprecedented levels of collaboration and funding.

Nevertheless, in response to HPCC and successive initiatives, the two newer subject areas of "computational science" [Stevenson 1994] and "net-centric computing" [ACM/IEEE 2001] have established themselves among the 11 that characterize computer science at this early moment in the 21st century. This Handbook views "computational science" as the application of computational and mathematical models and methods to science, having as a driving force the fundamental interaction between computation and scientific research. For instance, fields like computational astrophysics, computational biology,

and computational chemistry all unify the application of computing in science and engineering with underlying mathematical concepts, algorithms, graphics, and computer architecture. Much of the research and accomplishments of the computational science field is presented in Section III.

Net-centric computing, on the other hand, emphasizes the interactions among people, computers, and the Internet. It affects information technology systems in professional and personal spheres, including the implementation and use of search engines, commercial databases, and digital libraries, along with their risks and human factors. Some of these topics intersect in major ways with those of human–computer interaction, while others fall more directly in the realm of management information systems (MIS). Because MIS is widely viewed as a separate discipline from computer science, this Handbook does not attempt to cover all of MIS. However, it does address many MIS concerns in Section V (human–computer interaction) Section VI (information management), and Section VIII (net-centric computing).

The remaining sections of this Handbook cover relatively traditional areas of computer science — algorithms and complexity, computer architecture, operating systems, programming languages, artificial intelligence, software engineering, and computer graphics. A more careful summary of these sections appears below.

1.5 Organization and Content

In the 1940s, computer science was identified with number crunching, and numerical analysis was considered a central tool. Hardware, logical design, and information theory emerged as important subfields in the early 1950s. Software and programming emerged as important subfields in the mid-1950s and soon dominated hardware as topics of study in computer science. In the 1960s, computer science could be comfortably classified into theory, systems (including hardware and software), and applications. Software engineering emerged as an important subdiscipline in the late 1960s. The 1980 Computer Science and Engineering Research Study (COSERS) [Arden 1980] classified the discipline into nine subfields:

1. Numerical computation
2. Theory of computation
3. Hardware systems
4. Artificial intelligence
5. Programming languages
6. Operating systems
7. Database management systems
8. Software methodology
9. Applications

This Handbook's organization presents computer science in the following 11 sections, which are the subfields defined in [ACM/IEEE 2001].

1. Algorithms and complexity
2. Architecture and organization
3. Computational science
4. Graphics and visual computing
5. Human–computer interaction
6. Information management
7. Intelligent systems
8. Net-centric computing
9. Operating systems
10. Programming languages
11. Software engineering

This overall organization shares much in common with that of the 1980 COSERS study. That is, except for some minor renaming, we can read this list as a broadening of numerical analysis into computational science, and an addition of the new areas of human–computer interaction and graphics. The other areas appear in both classifications with some name changes (theory of computation has become algorithms and complexity, artificial intelligence has become intelligent systems, applications has become net-centric computing, hardware systems has evolved into architecture and networks, and database has evolved into information management). The overall similarity between the two lists suggests that the discipline of computer science has stabilized in the past 25 years.

However, although this high-level classification has remained stable, the content of each area has evolved dramatically. We examine below the scope of each area individually, along with the topics in each area that are emphasized in this Handbook.

1.5.1 Algorithms and Complexity

The subfield of algorithms and complexity is interpreted broadly to include core topics in the theory of computation as well as data structures and practical algorithm techniques. Its chapters provide a comprehensive overview that spans both theoretical and applied topics in the analysis of algorithms. Chapter 3 provides an overview of techniques of algorithm design like divide and conquer, dynamic programming, recurrence relations, and greedy heuristics, while Chapter 4 covers data structures both descriptively and in terms of their space–time complexity.

Chapter 5 examines topics in complexity like P vs. NP and NP-completeness, while Chapter 6 introduces the fundamental concepts of computability and undecidability and formal models such as Turing machines. Graph and network algorithms are treated in Chapter 7, and algebraic algorithms are the subject of Chapter 8.

The wide range of algorithm applications is presented in Chapter 9 through Chapter 15. Chapter 9 covers cryptographic algorithms, which have recently become very important in operating systems and network security applications. Chapter 10 covers algorithms for parallel computer architectures, Chapter 11 discusses algorithms for computational geometry, while Chapter 12 introduces the rich subject of randomized algorithms. Pattern matching and text compression algorithms are examined in Chapter 13, and genetic algorithms and their use in the biological sciences are introduced in Chapter 14. Chapter 15 concludes this section with a treatment of combinatorial optimization.

1.5.2 Architecture

Computer architecture is the design of efficient and effective computer hardware at all levels, from the most fundamental concerns of logic and circuit design to the broadest concerns of parallelism and high-performance computing. The chapters in Section II span these levels, providing a sampling of the principles, accomplishments, and challenges faced by modern computer architects.

Chapter 16 introduces the fundamentals of logic design components, including elementary circuits, Karnaugh maps, programmable array logic, circuit complexity and minimization issues, arithmetic processes, and speedup techniques. Chapter 17 focuses on processor design, including the fetch/execute instruction cycle, stack machines, CISC vs. RISC, and pipelining. The principles of memory design are covered in Chapter 18, while the architecture of buses and other interfaces is addressed in Chapter 19. Chapter 20 discusses the characteristics of input and output devices like the keyboard, display screens, and multimedia audio devices. Chapter 21 focuses on the architecture of secondary storage devices, especially disks.

Chapter 22 concerns the design of effective and efficient computer arithmetic units, while Chapter 23 extends the design horizon by considering various models of parallel architectures that enhance the performance of traditional serial architectures. Chapter 24 focuses on the relationship between computer architecture and networks, while Chapter 25 covers the strategies employed in the design of fault-tolerant and reliable computers.

1.5.3 Computational Science

The area of computational science unites computation, experimentation, and theory as three fundamental modes of scientific discovery. It uses scientific visualization, made possible by simulation and modeling, as a window into the analysis of physical, chemical, and biological phenomena and processes, providing a virtual microscope for inquiry at an unprecedented level of detail.

This section focuses on the challenges and opportunities offered by very high-speed clusters of computers and sophisticated graphical interfaces that aid scientific research and engineering design. Chapter 26 introduces the section by presenting the fundamental subjects of computational geometry and grid generation. The design of graphical models for scientific visualization of complex physical and biological phenomena is the subject of Chapter 27.

Each of the remaining chapters in this section covers the computational challenges and discoveries in a specific scientific or engineering field. Chapter 28 presents the computational aspects of structural mechanics, Chapter 29 summarizes progress in the area of computational electromagnetics, and Chapter 30 addresses computational modeling in the field of fluid dynamics. Chapter 31 addresses the grand challenge of computational ocean modeling. Computational chemistry is the subject of Chapter 32, while Chapter 33 addresses the computational dimensions of astrophysics. Chapter 34 closes this section with a discussion of the dramatic recent progress in computational biology.

1.5.4 Graphics and Visual Computing

Computer graphics is the study and realization of complex processes for representing physical and conceptual objects visually on a computer screen. These processes include the internal modeling of objects, rendering, projection, and motion. An overview of these processes and their interaction is presented in Chapter 35.

Fundamental to all graphics applications are the processes of modeling and rendering. Modeling is the design of an effective and efficient internal representation for geometric objects, which is the subject of Chapter 36 and Chapter 37. Rendering, the process of representing the objects in a three-dimensional scene on a two-dimensional screen, is discussed in Chapter 38. Among its special challenges are the elimination of hidden surfaces and the modeling of color, illumination, and shading.

The reconstruction of scanned and digitally photographed images is another important area of computer graphics. Sampling, filtering, reconstruction, and anti-aliasing are the focus of Chapter 39. The representation and control of motion, or animation, is another complex and important area of computer graphics. Its special challenges are presented in Chapter 40.

Chapter 41 discusses volume datasets, and Chapter 42 looks at the emerging field of virtual reality and its particular challenges for computer graphics. Chapter 43 concludes this section with a discussion of progress in the computer simulation of vision.

1.5.5 Human–Computer Interaction

This area, the study of how humans and computers interact, has the goal of improving the quality of such interaction and the effectiveness of those who use technology in the workplace. This includes the conception, design, implementation, risk analysis, and effects of user interfaces and tools on the people who use them.

Modeling the organizational environments in which technology users work is the subject of Chapter 44. Usability engineering is the focus of Chapter 45, while Chapter 46 covers task analysis and the design of functionality at the user interface. The influence of psychological preferences of users and programmers and the integration of these preferences into the design process is the subject of Chapter 47.

Specific devices, tools, and techniques for effective user-interface design form the basis for the next few chapters in this section. Lower-level concerns for the design of interface software technology are addressed in Chapter 48. The special challenges of integrating multimedia with user interaction are presented in Chapter 49. Computer-supported collaboration is the subject of Chapter 50, and the impact of international standards on the user interface design process is the main concern of Chapter 51.

1.5.6 Information Management

The subject area of information management addresses the general problem of storing large amounts of data in such a way that they are reliable, up-to-date, accessible, and efficiently retrieved. This problem is prominent in a wide range of applications in industry, government, and academic research. Availability of such data on the Internet and in forms other than text (e.g., CD, audio, and video) makes this problem increasingly complex.

At the foundation are the fundamental data models (relational, hierarchical, and object-oriented) discussed in Chapter 52. The conceptual, logical, and physical levels of designing a database for high performance in a particular application domain are discussed in Chapter 53.

A number of basic issues surround the effective design of database models and systems. These include choosing appropriate access methods (Chapter 54), optimizing database queries (Chapter 55), controlling concurrency (Chapter 56), and processing transactions (Chapter 57).

The design of databases for distributed and parallel systems is discussed in Chapter 58, while the design of hypertext and multimedia databases is the subject of Chapter 59. The contemporary issue of database security and privacy protection, in both stand-alone and networked environments, is the subject of Chapter 60.

1.5.7 Intelligent Systems

The field of intelligent systems, often called artificial intelligence (AI), studies systems that simulate human rational behavior in all its forms. Current efforts are aimed at constructing computational mechanisms that process visual data, understand speech and written language, control robot motion, and model physical and cognitive processes. Robotics is a complex field, drawing heavily from AI as well as other areas of science and engineering.

Artificial intelligence research uses a variety of distinct algorithms and models. These include fuzzy, temporal, and other logics, as described in Chapter 61. The related idea of qualitative modeling is discussed in Chapter 62, while the use of complex specialized search techniques that address the combinatorial explosion of alternatives in AI problems is the subject of Chapter 63. Chapter 64 addresses issues related to the mechanical understanding of spoken language.

Intelligent systems also include techniques for automated learning and planning. The use of decision trees and neural networks in learning and other areas is the subject of Chapter 65 and Chapter 66. Chapter 67 presents the rationale and uses of planning and scheduling models, while Chapter 68 contains a discussion of deductive learning. Chapter 69 addresses the challenges of modeling from the viewpoint of cognitive science, while Chapter 70 treats the challenges of decision making under uncertainty.

Chapter 71 concludes this section with a discussion of the principles and major results in the field of robotics: the design of effective devices that simulate mechanical, sensory, and intellectual functions of humans in specific task domains such as navigation and planning.

1.5.8 Net-Centric Computing

Extending system functionality across a networked environment has added an entirely new dimension to the traditional study and practice of computer science. Chapter 72 presents an overview of network organization and topologies, while Chapter 73 describes network routing protocols. Basic issues in network management are addressed in Chapter 74.

The special challenges of information retrieval and data mining from large databases and the Internet are addressed in Chapter 75. The important topic of data compression for internetwork transmission and archiving is covered in Chapter 76.

Modern computer networks, especially the Internet, must ensure system integrity in the event of inappropriate access, unexpected malfunction and breakdown, and violations of data and system security or individual privacy. Chapter 77 addresses the principles surrounding these security and privacy issues. A discussion of some specific malicious software and hacking events appears in Chapter 78. This section concludes with Chapter 79, which discusses protocols for user authentication, access control, and intrusion detection.

1.5.9 Operating Systems

An operating system is the software interface between the computer and its applications. This section covers operating system analysis, design, and performance, along with the special challenges for operating systems in a networked environment. Chapter 80 briefly traces the historical development of operating systems and introduces the fundamental terminology, including process scheduling, memory management, synchronization, I/O management, and distributed systems.

The "process" is a key unit of abstraction in operating system design. Chapter 81 discusses the dynamics of processes and threads. Strategies for process and device scheduling are presented in Chapter 82. The special requirements for operating systems in real-time and embedded system environments are treated in Chapter 83. Algorithms and techniques for process synchronization and interprocess communication are the subject of Chapter 84.

Memory and input/output device management is also a central concern of operating systems. Chapter 85 discusses the concept of virtual memory, from its early incarnations to its uses in present-day systems and networks. The different models and access methods for secondary storage and filesystems are covered in Chapter 86.

The influence of networked environments on the design of distributed operating systems is considered in Chapter 87. Distributed and multiprocessor scheduling are the focus in Chapter 88, while distributed file and memory systems are discussed in Chapter 89.

1.5.10 Programming Languages

This section examines the design of programming languages, including their paradigms, mechanisms for compiling and runtime management, and theoretical models, type systems, and semantics. Overall, this section provides a good balance between considerations of programming paradigms, implementation issues, and theoretical models.

Chapter 90 considers traditional language and implementation questions for imperative programming languages such as Fortran, C, and Ada. Chapter 91 examines object-oriented concepts such as classes, inheritance, encapsulation, and polymorphism, while Chapter 92 presents the view of functional programming, including lazy and eager evaluation. Chapter 93 considers declarative programming in the logic/constraint programming paradigm, while Chapter 94 covers the design and use of special purpose scripting languages. Chapter 95 considers the emergent paradigm of event-driven programming, while Chapter 96 covers issues regarding concurrent, distributed, and parallel programming models.

Type systems are the subject of Chapter 97, while Chapter 98 covers programming language semantics. Compilers and interpreters for sequential languages are considered in Chapter 99, while the issues surrounding runtime environments and memory management for compilers and interpreters are addressed in Chapter 100.

Brief summaries of the main features and applications of several contemporary languages appear in Appendix D, along with links to Web sites for more detailed information on these languages.

1.5.11 Software Engineering

The section on software engineering examines formal specification, design, verification and testing, project management, and other aspects of the software process. Chapter 101 introduces general software qualities such as maintainability, portability, and reuse that are needed for high-quality software systems, while Chapter 109 covers the general topic of software architecture.

Chapter 102 reviews specific models of the software life cycle such as the waterfall and spiral models. Chapter 106 considers a more formal treatment of software models, including formal specification languages.

Chapter 103 deals with the traditional design process, featuring a case study in top-down functional design. Chapter 104 considers the complementary strategy of object-oriented software design. Chapter 105

treats the subject of validation and testing, including risk and reliability issues. Chapter 107 deals with the use of rigorous techniques such as formal verification for quality assurance.

Chapter 108 considers techniques of software project management, including team formation, project scheduling, and evaluation, while Chapter 110 concludes this section with a treatment of specialized system development.

1.6 Conclusion

In 2002, the ACM celebrated its 55th anniversary. These five decades of computer science are characterized by dramatic growth and evolution. While it is safe to reaffirm that the field has attained a certain level of maturity, we surely cannot assume that it will remain unchanged for very long. Already, conferences are calling for new visions that will enable the discipline to continue its rapid evolution in response to the world's continuing demand for new technology and innovation.

This Handbook is designed to convey the modern spirit, accomplishments, and direction of computer science as we see it in 2003. It interweaves theory with practice, highlighting "best practices" in the field as well as emerging research directions. It provides today's answers to computational questions posed by professionals and researchers working in all 11 subject areas. Finally, it identifies key professional and social issues that lie at the intersection of the technical aspects of computer science and the people whose lives are impacted by such technology.

The future holds great promise for the next generations of computer scientists. These people will solve problems that have only recently been conceived, such as those suggested by the HPCC as "grand challenges." To address these problems in a way that benefits the world's citizenry will require substantial energy, commitment, and real investment on the part of institutions and professionals throughout the field. The challenges are great, and the solutions are not likely to be obvious.

References

ACM Curriculum Committee on Computer Science 1968. Curriculum 68: recommendations for the undergraduate program in computer science. *Commun. ACM*, 11(3):151–197, March.

ACM Curriculum Committee on Computer Science 1978. Curriculum 78: recommendations for the undergraduate program in computer science. *Commun. ACM*, 22(3):147–166, March.

ACM Task Force on the Core of Computer Science: Denning, P., Comer, D., Gries, D., Mulder, M., Tucker, A., and Young, P., 1988. *Computing as a Discipline*. Abridged version, *Commun. ACM*, Jan. 1989.

ACM/IEEE-CS Joint Curriculum Task Force. Computing Curricula 1991. ACM Press. Abridged version, *Commun. ACM*, June 1991, and *IEEE Comput.* Nov. 1991.

ACM/IEEE-CS Joint Task Force. Computing Curricula 2001: Computer Science Volume. ACM and IEEE Computer Society, December 2001, (http://www.acm.org/sigcse/cc2001).

Arden, B., Ed., 1980. *What Can be Automated?* Computer Science and Engineering Research (COSERS) Study. MIT Press, Boston, MA.

Bryant, R.E. and M.Y. Vardi, 2001. 2000–2001 Taulbee Survey: Hope for More Balance in Supply and Demand. Computing Research Assoc (http://www.cra.org).

CSNRCTB 1992. Computer Science and National Research Council Telecommunications Board. *Computing the Future: A Broader Agenda for Computer Science and Engineering*. National Academy Press, Washington, D.C.

Economist 1993. The computer industry: reboot system and start again. *Economist*, Feb. 27.

Gibbs, N. and A. Tucker 1986. A Model Curriculum for a Liberal Arts Degree in Computer Science. *Communications of the ACM*, March.

IEEE-CS 1976. Education Committee of the IEEE Computer Society. *A Curriculum in Computer Science and Engineering*. IEEE Pub. EH0119-8, Jan. 1977.

IEEE-CS 1983. Educational Activities Board. *The 1983 Model Program in Computer Science and Engineering.* Tech. Rep. 932. Computer Society of the IEEE, December.

NSF 2002. National Science Foundation. *Science and Engineering Indicators* (Vol. I and II), National Science Board, Arlington, VA.

NSF 2003a. National Science Foundation. *Budget Overview FY 2003* (http://www.nsf.gov/bfa/bud/fy2003/overview.htm).

NSF 2003b. National Science Foundation. *Revolutionizing Science and Engineering through Cyberinfrastructure*, report of the NSF Blue-Ribbon Advisory Panel on Cyberinfrastructure, January.

OST 1989. Office of Science and Technology. *The Federal High Performance Computing and Communication Program.* Executive Office of the President, Washington, D.C.

Rosser, J.B. et al. 1966. *Digital Computer Needs in Universities and Colleges*. Publ. 1233, National Academy of Sciences, National Research Council, Washington, D.C.

Stevenson, D.E. 1994. Science, computational science, and computer science. *Commun. ACM*, December.

2

Ethical Issues for Computer Scientists

Deborah G. Johnson
University of Virginia

Keith W. Miller
University of Illinois

2.1 Introduction: Why a Chapter on Ethical Issues?

Computers have had a powerful impact on our world and are destined to shape our future. This observation, now commonplace, is the starting point for any discussion of professionalism and ethics in computing. The work of computer scientists and engineers is part of the social, political, economic, and cultural world in which we live, and it affects many aspects of that world. Professionals who work with computers have special knowledge. That knowledge, when combined with computers, has significant power to change people's lives — by changing socio-technical systems; social, political and economic institutions; and social relationships.

In this chapter, we provide a perspective on the role of computer and engineering professionals and we examine the relationships and responsibilities that go with having and using computing expertise. In addition to the topic of professional ethics, we briefly discuss several of the social–ethical issues created or exacerbated by the increasing power of computers and information technology: privacy, property, risk and reliability, and globalization.

Computers, digital data, and telecommunications have changed work, travel, education, business, entertainment, government, and manufacturing. For example, work now increasingly involves sitting in front of a computer screen and using a keyboard to make things happen in a manufacturing process or to keep track of records. In the past, these same tasks would have involved physically lifting, pushing, and twisting or using pens, paper, and file cabinets. Changes such as these in the way we do things have, in turn, fundamentally changed who we are as individuals, communities, and nations. Some would argue, for example, that new kinds of communities (e.g., cyberspace on the Internet) are forming, individuals are developing new types of personal identities, and new forms of authority and control are taking hold as a result of this evolving technology.

Computer technology is shaped by social–cultural concepts, laws, the economy, and politics. These same concepts, laws, and institutions have been pressured, challenged, and modified by computer technology. Technological advances can antiquate laws, concepts, and traditions, compelling us to reinterpret and create new laws, concepts, and moral notions. Our attitudes about work and play, our values, and our laws and customs are deeply involved in technological change.

When it comes to the social–ethical issues surrounding computers, some have argued that the issues are not unique. All of the ethical issues raised by computer technology can, it is said, be classified and worked out using traditional *moral* concepts, distinctions, and theories. There is nothing new here in the sense that we can understand the new issues using traditional moral concepts, such as privacy, property, and responsibility, and traditional moral values, such as individual freedom, autonomy, accountability, and community. These concepts and values predate computers; hence, it would seem there is nothing unique about *computer ethics*.

On the other hand, those who argue for the uniqueness of the issues point to the fundamental ways in which computers have changed so many human activities, such as manufacturing, record keeping, banking, international trade, education, and communication. Taken together, these changes are so radical, it is claimed, that traditional moral concepts, distinctions, and theories, if not abandoned, must be significantly reinterpreted and extended. For example, they must be extended to computer-mediated relationships, computer software, computer art, datamining, virtual systems, and so on.

The uniqueness of the ethical issues surrounding computers can be argued in a variety of ways. Computer technology makes possible a scale of activities not possible before. This includes a larger scale of record keeping of personal information, as well as larger-scale calculations which, in turn, allow us to build and do things not possible before, such as undertaking space travel and operating a global communication system. Among other things, the increased scale means finer-grained personal information collection and more precise data matching and datamining. In addition to scale, computer technology has involved the creation of new kinds of entities for which no rules initially existed: entities such as computer files, computer programs, the Internet, Web browsers, cookies, and so on. The uniqueness argument can also be made in terms of the power and pervasiveness of computer technology. Computers and information technology seem to be bringing about a magnitude of change comparable to that which took place during the Industrial Revolution, transforming our social, economic, and political institutions; our understanding of what it means to be human; and the distribution of power in the world. Hence, it would seem that the issues are at least special, if not unique.

In this chapter, we will take an approach that synthesizes these two views of computer ethics by assuming that the analysis of computer ethical issues involves both working on something new and drawing on something old. We will view issues in computer ethics as new species of older ethical problems [Johnson 1994], such that the issues can be understood using traditional moral concepts such as autonomy, privacy, property, and responsibility, while at the same time recognizing that these concepts may have to be extended to what is new and special about computers and the situations they create.

Most ethical issues arising around computers occur in contexts in which there are already social, ethical, and legal norms. In these contexts, often there are implicit, if not formal (legal), rules about how individuals are to behave; there are familiar practices, social meanings, interdependencies, and so on. In this respect, the issues are not new or unique, or at least cannot be resolved without understanding the prevailing context, meanings, and values. At the same time, the situation may have special features because of the involvement of computers — features that have not yet been addressed by prevailing norms. These features can make a moral difference. For example, although property rights and even intellectual property rights had been worked out long before the creation of software, when software first appeared, it raised a new form of property issue. Should the arrangement of icons appearing on the screen of a user interface be ownable? Is there anything intrinsically wrong in copying software? Software has features that make the distinction between idea and expression (a distinction at the core of copyright law) almost incoherent. As well, software has features that make standard intellectual property laws difficult to enforce. Hence, questions about what should be owned when it comes to software and how to evaluate violations of software ownership rights are not new in the sense that they are property rights issues, but they are new

in the sense that nothing with the characteristics of software had been addressed before. We have, then, a new species of traditional property rights.

Similarly, although our understanding of rights and responsibilities in the employer–employee relationship has been evolving for centuries, never before have employers had the capacity to monitor their workers electronically, keeping track of every keystroke, and recording and reviewing all work done by an employee (covertly or with prior consent). When we evaluate this new monitoring capability and ask whether employers should use it, we are working on an issue that has never arisen before, although many other issues involving employer–employee rights have. We must address a new species of the tension between employer–employee rights and interests.

The social–ethical issues posed by computer technology are significant in their own right, but they are of special interest here because computer and engineering professionals bear responsibility for this technology. It is of critical importance that they understand the social change brought about by their work and the difficult social–ethical issues posed. Just as some have argued that the social–ethical issues posed by computer technology are not unique, some have argued that the issues of professional ethics surrounding computers are not unique. We propose, in parallel with our previous genus–species account, that the professional ethics issues arising for computer scientists and engineers are species of generic issues of professional ethics. All professionals have responsibilities to their employers, clients, co-professionals, and the public. Managing these types of responsibilities poses a challenge in all professions. Moreover, all professionals bear some responsibility for the impact of their work. In this sense, the professional ethics issues arising for computer scientists and engineers are generally similar to those in other professions. Nevertheless, it is also true to say that the issues arise in unique ways for computer scientists and engineers because of the special features of computer technology.

In what follows, we discuss ethics in general, professional ethics, and finally, the ethical issues surrounding computer and information technology.

2.2 Ethics in General

Rigorous study of ethics has traditionally been the purview of philosophers and scholars of religious studies. Scholars of ethics have developed a variety of ethical theories with several tasks in mind:

To explain and justify the idea of morality and prevailing moral notions
To critique ordinary moral beliefs
To assist in rational, ethical decision making

Our aim in this chapter is not to propose, defend, or attack any particular ethical theory. Rather, we offer brief descriptions of three major and influential ethical theories to illustrate the nature of ethical analysis. We also include a decision-making method that combines elements of each theory.

Ethical analysis involves giving reasons for moral claims and commitments. It is not just a matter of articulating intuitions. When the reasons given for a claim are developed into a moral theory, the theory can be incorporated into techniques for improved technical decision making. The three ethical theories described in this section represent three traditions in ethical analysis and problem solving. The account we give is not exhaustive, nor is our description of the three theories any more than a brief introduction. The three traditions are utilitarianism, deontology, and social contract theory.

2.2.1 Utilitarianism

Utilitarianism has greatly influenced 20th-century thinking, especially insofar as it influenced the development of cost–benefit analysis. According to utilitarianism, we should make decisions about what to do by focusing on the consequences of actions and policies; we should choose actions and policies that bring about the best consequences. Ethical rules are derived from their usefulness (their utility) in bringing about happiness. In this way, utilitarianism offers a seemingly simple moral principle to determine what to do

in a given situation: everyone ought to act so as to bring about the greatest amount of happiness for the greatest number of people.

According to utilitarianism, happiness is the only value that can serve as a foundational base for ethics. Because happiness is the ultimate good, morality must be based on creating as much of this good as possible. The utilitarian principle provides a decision procedure. When you want to know what to do, the right action is the alternative that produces the most overall net happiness (happiness-producing consequences minus unhappiness-producing consequences). The right action may be one that brings about some unhappiness, but that is justified if the action also brings about enough happiness to counterbalance the unhappiness or if the action brings about the least unhappiness of all possible alternatives.

Utilitarianism should not be confused with egoism. Egoism is a theory claiming that one should act so as to bring about the most good consequences *for oneself*. Utilitarianism does not say that you should maximize your own good. Rather, total happiness in the world is what is at issue; when you evaluate your alternatives, you must ask about their effects on the happiness of everyone. It may turn out to be right for you to do something that will diminish your own happiness because it will bring about an increase in overall happiness.

The emphasis on consequences found in utilitarianism is very much a part of personal and policy decision making in our society, in particular as a framework for law and public policy. Cost–benefit and risk–benefit analysis are, for example, consequentialist in character.

Utilitarians do not all agree on the details of utilitarianism; there are different kinds of utilitarianism. One issue is whether the focus should be on *rules* of behavior or individual *acts*. Utilitarians have recognized that it would be counter to overall happiness if each one of us had to calculate at every moment what the consequences of every one of our actions would be. Sometimes we must act quickly, and often the consequences are difficult or impossible to foresee. Thus, there is a need for general rules to guide our actions in ordinary situations. Hence, *rule-utilitarians* argue that we ought to adopt rules that, if followed by everyone, would, in general and in the long run, maximize happiness. *Act-utilitarians*, on the other hand, put the emphasis on judging individual actions rather than creating rules.

Both rule-utilitarians and act-utilitarians, nevertheless, share an emphasis on consequences; deontological theories do not share this emphasis.

2.2.2 Deontological Theories

Deontological theories can be understood as a response to important criticisms of utilitarian theories. A standard criticism is that utilitarianism seems to lead to conclusions that are incompatible with our most strongly held moral intuitions. Utilitarianism seems, for example, open to the possibility of justifying enormous burdens on some individuals for the sake of others. To be sure, every person counts equally; no one person's happiness or unhappiness is more important than any other person's. However, because utilitarians are concerned with the total amount of happiness, we can imagine situations where great overall happiness would result from sacrificing the happiness of a few. Suppose, for example, that having a small number of slaves would create great happiness for large numbers of people; or suppose we kill one healthy person and use his or her body parts to save ten people in need of transplants.

Critics of utilitarianism say that if utilitarianism justifies such practices, then the theory must be wrong. Utilitarians have a defense, arguing that such practices could not be justified in utilitarianism because of the long-term consequences. Such practices would produce so much fear that the happiness temporarily created would never counterbalance the unhappiness of everyone living in fear that they might be sacrificed for the sake of overall happiness.

We need not debate utilitarianism here. The point is that deontologists find utilitarianism problematic because it puts the emphasis on the consequences of an act rather than on the quality of the act itself. Deontological theories claim that the internal character of the act is what is important. The rightness or wrongness of an action depends on the principles inherent in the action. If an action is done from a sense of duty, and if the principle of the action can be universalized, then the action is right. For example, if I tell the truth because it is convenient for me to do so or because I fear the consequences of getting caught in a

lie, my action is not worthy. A worthy action is an action that is done from duty, which involves respecting other people and recognizing them as ends in themselves, not as means to some good effect.

According to deontologists, utilitarianism is wrong because it treats individuals as means to an end (maximum happiness). For deontologists, what grounds morality is not happiness, but human beings as rational agents. Human beings are capable of reasoning about what they want to do. The laws of nature determine most activities: plants grow toward the sun, water boils at a certain temperature, and objects accelerate at a constant rate in a vacuum. Human action is different in that it is self-determining; humans initiate action after thinking, reasoning, and deciding. The human capacity for rational decisions makes morality possible, and it grounds deontological theory. Because each human being has this capacity, each human being must be treated accordingly — with respect. No one else can make our moral choices for us, and each of us must recognize this capacity in others.

Although deontological theories can be formulated in a number of ways, one formulation is particularly important: Immanuel Kant's categorical imperative [Kant 1785]. There are three versions of it, and the second version goes as follows: *Never treat another human being merely as a means but always as an end*. It is important to note the *merely* in the categorical imperative. Deontologists do not insist that we never use another person; only that we never *merely* use them. For example, if I own a company and hire employees to work in my company, I might be thought of as using those employees as a means to my end (i.e., the success of my business). This, however, is not wrong if the employees agree to work for me and if I pay them a fair wage. I thereby respect their ability to choose for themselves, and I respect the value of their labor. What would be wrong would be to take them as slaves and make them work for me, or to pay them so little that they must borrow from me and remain always in my debt. This would show disregard for the value of each person as a freely choosing, rationally valuing, efficacious person.

2.2.3 Social Contract Theories

A third tradition in ethics thinks of ethics on the model of a social contract. There are many different social contract theories, and some, at least, are based on a deontological principle. Individuals are rational free agents; hence, it is immoral to exert undue power over them, that is, to coerce them. Government and society are problematic insofar as they seem to force individuals to obey rules, apparently treating individuals as means to social good. Social contract theories get around this problem by claiming that morality (and government policy) is, in effect, the outcome of rational agents agreeing to social rules. In agreeing to live by certain rules, we make a contract. Morality and government are not, then, systems imposed on individuals; they do not exactly involve coercion. Rather, they are systems created by freely choosing individuals (or they are institutions that rational individuals would choose if given the opportunity).

Philosophers such as Rousseau, Locke, Hobbes, and more recently Rawls [1971] are generally considered social contract theorists. They differ in how they get to the social contract and what it implies. For our purposes, however, the key idea is that principles and rules guiding behavior may be derived from identifying what it is that rational (even self-interested) individuals would agree to in making a social contract. Such principles and rules are the basis of a shared morality. For example, it would be rational for me to agree to live by rules that forbid killing and lying. Even though such rules constrain me, they also give me some degree of protection: if they are followed, I will not be killed or lied to.

It is important to note, however, that social contract theory cannot be used simply by asking what rules you would agree to *now*. Most theorists recognize that what you would agree to now is influenced by your present position in society. Most individuals would opt for rules that would benefit their particular situation and characteristics. Hence, most social contract theorists insist that the principles or rules of the social contract must be derived by assuming certain things about human nature or the human condition. Rawls, for example, insists that we imagine ourselves behind a *veil of ignorance*. We are not allowed to know important features about ourselves (e.g., what talents we have, what race or gender we are), for if we know these things, we will not agree to just rules, but only to rules that will maximize our self-interest. Justice consists of the rules we would agree to when we do not know who we are, for we would want rules that would give us a fair situation no matter where we ended up in the society.

2.2.4 A Paramedic Method for Computer Ethics

Drawing on elements of the three theories described, Collins and Miller [1992] have proposed a decision-assisting method, called the *paramedic method for computer ethics*. This is *not* an algorithm for solving ethical problems; it is not nearly detailed or objective enough for that designation. It is merely a guideline for an organized approach to ethical problem solving.

Assume that a computer professional is faced with a decision that involves human values in a significant way. There may already be some obvious alternatives, and there also may be creative solutions not yet discovered. The paramedic method is designed to help the professional to analyze alternative actions and to encourage the development of creative solutions. To illustrate the method, suppose you are in a tight spot and do not know exactly what the right thing to do is. The method proceeds as follows:

1. Identify alternative actions; list the few alternatives that seem most promising. If an action requires a long description, summarize it as a title with just a few words. Call the alternative actions A_1, $A_2, \ldots, A_a$. No more than five actions should be analyzed at a time.
2. Identify people, groups of people, or organizations that will be affected by each of the alternative decision-actions. Again, hold down the number of entities to the five or six that are affected most. Label the people $P_1, P_2, \ldots, P_p$.
3. Make a table with the horizontal rows labeled by the identified people and the vertical columns labeled with the identified actions. We call such a table a *P × A table*. Make two copies of the P × A table; label one the *opportunities* table and the other the *vulnerabilities* table. In the opportunities table, list in each interior cell of the table at entry $[x, y]$ the possible good that is likely to happen to person x if action y is taken. Similarly, in the vulnerability table, at position $[x, y]$ list all of the things that are likely to happen badly for x if the action y is taken. These two graphs represent benefit–cost calculations for a consequentialist, utilitarian analysis.
4. Make a new table with the set of persons marking both the columns and the rows (a P × P table). In each cell $[x, y]$ name any responsibilities or duties that x owes y in this situation. (The cells on the diagonal $[x, x]$ are important; they list things one owes oneself.) Now, make copies of this table, labeling one copy for each of the alternative actions being considered. Work through each cell $[x, y]$ of each table and place a + next to a duty if the action for that table is likely to fulfill the duty x owes y; mark the duty with a − if the action is unlikely to fulfill that duty; mark the duty with a +/− if the action partially fulfills it and partially does not; and mark the duty with a ? if the action is irrelevant to the duty or if it is impossible to predict whether or not the duty will be fulfilled. (Few cells generally fall into this last category.)
5. Review the tables from steps 3 and 4. Envision a meeting of all of the parties (or one representative from each of the groups) in which no one knows which role they will take or when they will leave the negotiation. Which alternative do you think such a group would adopt, if any? Do you think such a group could discover a new alternative, perhaps combining the best elements of the previously listed actions? If this thought experiment produces a new alternative, expand the P × A tables from step 3 to include the new alternative action, make a new copy of the P × P table in step 4, and do the + and − marking for the new table.
6. If any one of the alternatives seems to be clearly preferred (i.e., it has high opportunity and low vulnerability for all parties and tends to fulfill all the duties in the P × P table), then that becomes the recommended decision. If no one alternative action stands out, the professionals can examine trade-offs using the charts or can iteratively attempt step 5 (perhaps with outside consultations) until an acceptable alternative is generated.

Using the paramedic method can be time consuming, and it does not eliminate the need for judgment. But it can help organize and focus analysis as an individual or a group works through the details of a situation to arrive at a decision.

2.2.5 Easy and Hard Ethical Decision Making

Sometimes ethical decision making is easy; for example, when it is clear that an action will prevent a serious harm and has no drawbacks, then that action is the right thing to do. Sometimes, however, ethical decision making is more complicated and challenging. Take the following case: your job is to make decisions about which parts to buy for a computer manufacturing company. A person who sells parts to the company offers you tickets to an expensive Broadway show. Should you accept the tickets? In this case, the right thing to do is more complicated because you may be able to accept the tickets and not have this affect your decision about parts. You owe your employer a decision on parts that is in the best interests of the company, but will accepting the tickets influence future decisions?

Other times, you know what the right thing to do is, but doing it will have such great personal costs that you cannot bring yourself to do it. For example, you might be considering blowing the whistle on your employer, who has been extremely kind and generous to you, but who now has asked you to cheat on the testing results on a life-critical software system designed for a client.

To make good decisions, professionals must be aware of potential issues and must have a fairly clear sense of their responsibilities in various kinds of situations. This often requires sorting out complex relationships and obligations, anticipating the effects of various actions, and balancing responsibilities to multiple parties. This activity is part of professional ethics.

2.3 Professional Ethics

Ethics is not just a matter for individuals as individuals. We all occupy a variety of social roles that involve special responsibilities and privileges. As parents, we have special responsibilities for children. As citizens, members of churches, officials in clubs, and so on, we have special rights and duties — and so it is with professional roles. Being a professional is often distinguished from merely having an occupation, because a professional makes a different sort of commitment. Being a professional means more than just having a job. The difference is commitment to doing the right thing because you are a member of a group that has taken on responsibility for a domain of activity. The group is accountable to society for this domain, and for this reason, professionals must behave in ways that are worthy of public trust.

Some theorists explain this commitment in terms of a social contract between a profession and the society in which it functions. Society grants special rights and privileges to the professional group, such as control of admission to the group, access to educational institutions, and confidentiality in professional–client relationships. Society, in turn, may even grant the group a monopoly over a domain of activity (e.g., only licensed engineers can sign off on construction designs, and only doctors can prescribe drugs). In exchange, the professional group promises to self-regulate and practice its profession in ways that are beneficial to society, that is, to promote safety, health, and welfare. The social contract idea is a way of illustrating the importance of the trust that clients and the public put in professionals; it shows the importance of professionals acting so as to be worthy of that trust.

The special responsibilities of professionals have been accounted for in other theoretical frameworks, as well. For example, Davis [1995] argues that members of professions implicitly, if not explicitly, agree among themselves to adhere to certain standards because this elevates the level of activity. If all computer scientists and engineers, for example, agreed never to release software that has not met certain testing standards, this would prevent market pressures from driving down the quality of software being produced. Davis's point is that the special responsibilities of professionals are grounded in what members of a professional group owe to one another: they owe it to one another to live up to agreed-upon rules and standards. Other theorists have tried to ground the special responsibilities of professionals in ordinary morality. Alpern [1991] argues, for example, that the engineer's responsibility for safety derives from the ordinary moral edict *do no harm*. Because engineers are in a position to do greater harm than others, engineers have a special responsibility in their work to take greater care.

In the case of computing professionals, responsibilities are not always well articulated because of several factors. Computing is a relatively new field. There is no single unifying professional association that

controls membership, specifies standards of practice, and defines what it means to be a member of the profession. Moreover, many computer scientists and engineers are employees of companies or government agencies, and their role as computer professional may be somewhat in tension with their role as an employee of the company or agency. This can blur an individual's understanding of his or her professional responsibilities. Being a professional means having the independence to make decisions on the basis of special expertise, but being an employee often means acting in the best interests of the company, i.e., being loyal to the organization. Another difficulty in the role of computing professional is the diversity of the field. Computing professionals are employed in a wide variety of contexts, have a wide variety of kinds of expertise, and come from diverse educational backgrounds. As mentioned before, there is no single unifying organization, no uniform admission standard, and no single identifiable professional role.

To be sure, there are pressures on the field to move more in the direction of professionalization, but this seems to be happening to factions of the group rather than to the field as a whole. An important event moving the field in the direction of professionalization was the decision of the state of Texas to provide a licensing system for software engineers. The system specifies a set of requirements and offers an exam that must be passed in order for a computer professional to receive a software engineering license.

At the moment, Texas is the only state that offers such a license, so the field of computing remains loosely organized. It is not a strongly differentiated profession in the sense that there is no single characteristic (or set of characteristics) possessed by all computer professionals, no characteristic that distinguishes members of the group from anyone who possesses knowledge of computing. At this point, the field of computing is best described as a large group of individuals, all of whom work with computers, many of whom have expertise in subfields; they have diverse educational backgrounds, follow diverse career paths, and engage in a wide variety of job activities.

Despite the lack of unity in the field, there are many professional organizations, several professional codes of conduct, and expectations for professional practice. The codes of conduct, in particular, form the basis of an emerging professional ethic that may, in the future, be refined to the point where there will be a strongly differentiated role for computer professionals.

Professional codes play an important role in articulating a collective sense of both the ideal of the profession and the minimum standards required. Codes of conduct state the consensus views of members while shaping behavior.

A number of professional organizations have codes of ethics that are of interest here. The best known include the following:

- The Association for Computing Machinery (ACM) Code of Ethics and Professional Conduct (see Appendix B)
- The Institute of Electrical and Electronic Engineers (IEEE) Code of Ethics
- The Joint ACM/IEEE Software Engineering Code of Ethics and Professional Practice
- The Data Processing Managers Association (DPMA, now the Association of Information Technology Professionals [AITP]) Code of Ethics and Standards of Conduct
- The Institute for Certification of Computer Professionals (ICCP) Code of Ethics
- The Canadian Information Processing Society Code of Ethics
- The British Computer Society Code of Conduct

Each of these codes has different emphases and goals. Each in its own way, however, deals with issues that arise in the context in which computer scientists and engineers typically practice.

The codes are relatively consistent in identifying computer professionals as having responsibilities to be faithful to their employers and clients, and to protect public safety and welfare. The most salient ethical issues that arise in professional practice have to do with balancing these responsibilities with personal (or nonprofessional) responsibilities. Two common areas of tension are worth mentioning here, albeit briefly.

As previously mentioned, computer scientists may find themselves in situations in which their responsibility as professionals to protect the public comes into conflict with loyalty to their employer. Such situations sometimes escalate to the point where the computer professional must decide whether to blow

the whistle. Such a situation might arise, for example, when the computer professional believes that a piece of software has not been tested enough but her employer wants to deliver the software on time and within the allocated budget (which means immediate release and no more resources being spent on the project). Whether to blow the whistle is one of the most difficult decisions computer engineers and scientists may have to face. Whistle blowing has received a good deal of attention in the popular press and in the literature on professional ethics, because this tension seems to be built into the role of engineers and scientists, that is, the combination of being a professional with highly technical knowledge and being an employee of a company or agency.

Of course, much of the literature on whistle blowing emphasizes strategies that avoid the need for it. Whistle blowing can be avoided when companies adopt mechanisms that give employees the opportunity to express their concerns without fear of repercussions, for example, through ombudspersons to whom engineers and scientists can report their concerns anonymously. The need to blow the whistle can also be diminished when professional societies maintain hotlines that professionals can call for advice on how to get their concerns addressed.

Another important professional ethics issue that often arises is directly tied to the importance of being worthy of client (and, indirectly, public) trust. Professionals can find themselves in situations in which they have (or are likely to have) a conflict of interest. A conflict-of-interest situation is one in which the professional is hired to perform work for a client and the professional has some personal or professional interest that may (or may appear to) interfere with his or her judgment on behalf of the client. For example, suppose a computer professional is hired by a company to evaluate its needs and recommend hardware and software that will best suit the company. The computer professional does precisely what is requested, but fails to mention being a silent partner in a company that manufactures the hardware and software that has been recommended. In other words, the professional has a personal interest — financial benefit — in the company's buying certain equipment. If the company were told this upfront, it might expect the computer professional to favor his own company's equipment; however, if the company finds out about the affiliation later on, it might rightly think that it had been deceived. The professional was hired to evaluate the needs of the company and to determine how best to meet those needs, and in so doing to have the best interests of the company fully in mind. Now, the company suspects that the professional's judgment was biased. The professional had an interest that might have interfered with his judgment on behalf of the company.

There are a number of strategies that professions use to avoid these situations. A code of conduct may, for example, specify that professionals reveal all relevant interests to their clients before they accept a job. Or the code might specify that members never work in a situation where there is even the appearance of a conflict of interest.

This brings us to the special character of computer technology and the effects that the work of computer professionals can have on the shape of the world. Some may argue that computer professionals have very little say in what technologies get designed and built. This seems to be mistaken on at least two counts. First, we can distinguish between computer professionals as individuals and computer professionals as a group. Even if individuals have little power in the jobs they hold, they can exert power collectively. Second, individuals can have an effect if they think of themselves as professionals and consider it their responsibility to anticipate the impact of their work.

2.4 Ethical Issues That Arise from Computer Technology

The effects of a new technology on society can draw attention to an old issue and can change our understanding of that issue. The issues listed in this section — privacy, property rights, risk and reliability, and global communication — were of concern, even problematic, before computers were an important technology. But computing and, more generally, electronic telecommunications, have added new twists and new intensity to each of these issues. Although computer professionals cannot be expected to be experts on all of these issues, it is important for them to understand that computer technology is shaping the world. And it is important for them to keep these impacts in mind as they work with computer technology. Those

who are aware of privacy issues, for example, are more likely to take those issues into account when they design database management systems; those who are aware of risk and reliability issues are more likely to articulate these issues to clients and attend to them in design and documentation.

2.4.1 Privacy

Privacy is a central topic in computer ethics. Some have even suggested that privacy is a notion that has been antiquated by technology and that it should be replaced by a new openness. Others think that computers must be harnessed to help restore as much privacy as possible to our society. Although they may not like it, computer professionals are at the center of this controversy. Some are designers of the systems that facilitate information gathering and manipulation; others maintain and protect the information. As the saying goes, *information is power* — but power can be used or abused.

Computer technology creates wide-ranging possibilities for tracking and monitoring of human behavior. Consider just two ways in which personal privacy may be affected by computer technology. First, because of the capacity of computers, massive amounts of information can be gathered by record-keeping organizations such as banks, insurance companies, government agencies, and educational institutions. The information gathered can be kept and used indefinitely, and shared with other organizations rapidly and frequently. A second way in which computers have enhanced the possibilities for monitoring and tracking of individuals is by making possible new kinds of information. When activities are done using a computer, transactional information is created. When individuals use automated bank teller machines, records are created; when certain software is operating, keystrokes on a computer keyboard are recorded; the content and destination of electronic mail can be tracked, and so on. With the assistance of newer technologies, much more of this transactional information is likely to be created. For example, television advertisers may be able to monitor television watchers with scanning devices that record who is sitting in a room facing the television. Highway systems allow drivers to pass through toll booths without stopping as a beam reading a bar code on the automobile charges the toll, simultaneously creating a record of individual travel patterns. All of this information (transactional and otherwise) can be brought together to create a detailed portrait of a person's life, a portrait that the individual may never see, although it is used by others to make decisions about the individual.

This picture suggests that computer technology poses a serious threat to personal privacy. However, one can counter this picture in a number of ways. Is it computer technology *per se* that poses the threat or is it just the way the technology has been used (and is likely to be used in the future)? Computer professionals might argue that they create the technology but are not responsible for how it is used. This argument is, however, problematic for a number of reasons and perhaps foremost because it fails to recognize the potential for solving some of the problems of abuse in the design of the technology. Computer professionals are in the ideal position to think about the potential problems with computers and to design so as to avoid these problems. When, instead of deflecting concerns about privacy as out of their purview, computer professionals set their minds to solve privacy and security problems, the systems they design can improve.

At the same time we think about changing computer technology, we also must ask deeper questions about privacy itself and what it is that individuals need, want, or are entitled to when they express concerns about the loss of privacy. In this sense, computers and privacy issues are ethical issues. They compel us to ask deep questions about what makes for a good and just society. Should individuals have more choice about who has what information about them? What is the proper relationship between citizens and government, between individuals and private corporations? How are we to negotiate the tension between the competing needs for privacy and security? As previously suggested, the questions are not completely new, but some of the possibilities created by computers are new, and these possibilities do not readily fit the concepts and frameworks used in the past. Although we cannot expect computer professionals to be experts on the philosophical and political analysis of privacy, it seems clear that the more they know, the better the computer technology they produce is likely to be.

2.4.2 Property Rights and Computing

The protection of intellectual property rights has become an active legal and ethical debate, involving national and international players. Should software be copyrighted, patented, or free? Is computer software a process, a creative work, a mathematical formalism, an idea, or some combination of these? What is society's stake in protecting software rights? What is society's stake in widely disseminating software? How do corporations and other institutions protect their rights to ideas developed by individuals? And what are the individuals' rights? Such questions must be answered publicly through legislation, through corporate policies, and with the advice of computing professionals. Some of the answers will involve technical details, and all should be informed by ethical analysis and debate.

An issue that has received a great deal of legal and public attention is the ownership of software. In the course of history, software is a relatively new entity. Whereas Western legal systems have developed property laws that encourage invention by granting certain rights to inventors, there are provisions against ownership of things that might interfere with the development of the technological arts and sciences. For this reason, copyrights protect only the expression of ideas, not the ideas themselves, and we do not grant patents on laws of nature, mathematical formulas, and abstract ideas. The problem with computer software is that it has not been clear that we could grant ownership of it without, in effect, granting ownership of numerical sequences or mental steps. Software can be copyrighted, because a copyright gives the holder ownership of the *expression* of the idea (not the idea itself), but this does not give software inventors as much protection as they need to compete *fairly*. Competitors may see the software, grasp the idea, and write a somewhat different program to do the same thing. The competitor can sell the software at less cost because the cost of developing the first software does not have to be paid. Patenting would provide stronger protection, but until quite recently the courts have been reluctant to grant this protection because of the problem previously mentioned: patents on software would appear to give the holder control of the building blocks of the technology, an ownership comparable to owning ideas themselves. In other words, too many patents may interfere with technological development.

Like the questions surrounding privacy, property rights in computer software also lead back to broader ethical and philosophical questions about what constitutes a just society. In computing, as in other areas of technology, we want a system of property rights that promotes invention (creativity, progress), but at the same time, we want a system that is fair in the sense that it rewards those who make significant contributions but does not give anyone so much control that others are prevented from creating. Policies with regard to property rights in computer software cannot be made without an understanding of the technology. This is why it is so important for computer professionals to be involved in public discussion and policy setting on this topic.

2.4.3 Risk, Reliability, and Accountability

As computer technology becomes more important to the way we live, its risks become more worrisome. System errors can lead to physical danger, sometimes catastrophic in scale. There are security risks due to hackers and crackers. Unreliable data and intentional misinformation are risks that are increased because of the technical and economic characteristics of digital data. Furthermore, the use of computer programs is, in a practical sense, inherently unreliable.

Each of these issues (and many more) requires computer professionals to face the linked problems of risk, reliability, and accountability. Professionals must be candid about the risks of a particular application or system. Computing professionals should take the lead in educating customers and the public about what predictions we can and cannot make about software and hardware reliability. Computer professionals should make realistic assessments about costs and benefits, and be willing to take on both for projects in which they are involved.

There are also issues of sharing risks as well as resources. Should liability fall to the individual who buys software or to the corporation that developed it? Should society acknowledge the inherent risks in using

software in life-critical situations and shoulder some of the responsibility when something goes wrong? Or should software providers (both individuals and institutions) be exclusively responsible for software safety? All of these issues require us to look at the interaction of technical decisions, human consequences, rights, and responsibilities. They call not just for technical solutions but for solutions that recognize the kind of society we want to have and the values we want to preserve.

2.4.4 Rapidly Evolving Globally Networked Telecommunications

The system of computers and connections known as the Internet provides the infrastructure for new kinds of communities — electronic communities. Questions of individual accountability and social control, as well as matters of etiquette, arise in electronic communities, as in all societies. It is not just that we have societies forming in a new physical environment; it is also that ongoing electronic communication changes the way individuals understand their identity, their values, and their plans for their lives. The changes that are taking place must be examined and understood, especially the changes affecting fundamental social values such as democracy, community, freedom, and peace.

Of course, speculating about the Internet is now a popular pastime, and it is important to separate the hype from the reality. The reality is generally much more complex and much more subtle. We will not engage in speculation and prediction about the future. Rather, we want to emphasize how much better off the world would be if (instead of watching social impacts of computer technology after the fact) computer engineers and scientists were thinking about the potential effects early in the design process. Of course, this can only happen if computer scientists and engineers are encouraged to see the social–ethical issues as a component of their professional responsibility. This chapter has been written with that end in mind.

2.5 Final Thoughts

Computer technology will, no doubt, continue to evolve and will continue to affect the character of the world we live in. Computer scientists and engineers will play an important role in shaping the technology. The technologies we use shape how we live and who we are. They make every difference in the moral environment in which we live. Hence, it seems of utmost importance that computer scientists and engineers understand just how their work affects humans and human values.

References

Alpern, K.D. 1991. Moral responsibility for engineers. In *Ethical Issues in Engineering*, D.G. Johnson, Ed., pp. 187–195. Prentice Hall, Englewood Cliffs, NJ.

Collins, W.R., and Miller, K. 1992. A paramedic method for computing professionals. *J. Syst. Software*. 17(1): 47–84.

Davis, M. 1995. Thinking like an engineer: the place of a code of ethics in the practice of a profession. In *Computers, Ethics, and Social Values*, D.G. Johnson and H. Nissenbaum, Eds., pp. 586–597. Prentice Hall, Englewood Cliffs, NJ.

Johnson, D.G. 2001. *Computer Ethics*, *3rd edition*. Prentice Hall, Englewood Cliffs, NJ.

Kant, I. 1785. *Foundations of the Metaphysics of Morals*. L. Beck, trans., 1959. Library of Liberal Arts, 1959.

Rawls, J. 1971. *A Theory of Justice*. Harvard Univ. Press, Cambridge, MA.

I

Algorithms and Complexity

This section addresses the challenges of solving hard problems algorithmically and efficiently. These chapters cover basic methodologies (divide and conquer), data structures, complexity theory (space and time measures), parallel algorithms, and strategies for solving hard problems and identifying unsolvable problems. They also cover some exciting contemporary applications of algorithms, including cryptography, genetics, graphs and networks, pattern matching and text compression, and geometric and algebraic algorithms.

3

Basic Techniques for Design and Analysis of Algorithms

Edward M. Reingold
Illinois Institute of Technology

3.1 Introduction

We outline the basic methods of algorithm design and analysis that have found application in the manipulation of discrete objects such as lists, arrays, sets, graphs, and geometric objects such as points, lines, and polygons. We begin by discussing recurrence relations and their use in the analysis of algorithms. Then we discuss some specific examples in algorithm analysis, sorting, and priority queues. In the final three sections, we explore three important techniques of algorithm design: divide-and-conquer, dynamic programming, and greedy heuristics.

3.2 Analyzing Algorithms

It is convenient to classify algorithms based on the relative amount of time they require: how fast does the time required grow as the size of the problem increases? For example, in the case of arrays, the "size of the problem" is ordinarily the number of elements in the array. If the size of the problem is measured by a variable n, we can express the time required as a function of n, $T(n)$. When this function $T(n)$ grows rapidly, the algorithm becomes unusable for large n; conversely, when $T(n)$ grows slowly, the algorithm remains useful even when n becomes large.

We say an algorithm is $\Theta(n^2)$ if the time it takes quadruples when n doubles; an algorithm is $\Theta(n)$ if the time it takes doubles when n doubles; an algorithm is $\Theta(\log n)$ if the time it takes increases by a constant, independent of n, when n doubles; an algorithm is $\Theta(1)$ if its time does not increase at all when n increases. In general, an algorithm is $\Theta(T(n))$ if the time it requires on problems of size n grows proportionally to $T(n)$ as n increases. Table 3.1 summarizes the common growth rates encountered in the analysis of algorithms.

TABLE 3.1 Common Growth Rates of Times of Algorithms

Rate of Growth	Comment	Examples
$\Theta(1)$	Time required is constant, independent of problem size	Expected time for hash searching
$\Theta(\log\log n)$	Very slow growth of time required	Expected time of interpolation search
$\Theta(\log n)$	Logarithmic growth of time required — doubling the problem size increases the time by only a constant amount	Computing x^n; binary search of an array
$\Theta(n)$	Time grows linearly with problem size — doubling the problem size doubles the time required	Adding/subtracting n-digit numbers; linear search of an n-element array
$\Theta(n \log n)$	Time grows worse than linearly, but not much worse — doubling the problem size more than doubles the time required	Merge sort; heapsort; lower bound on comparison-based sorting
$\Theta(n^2)$	Time grows quadratically — doubling the problem size quardruples the time required	Simple-minded sorting algorithms
$\Theta(n^3)$	Time grows cubically — doubling the problem size results in an eight fold increase in the time required	Ordinary matrix multiplication
$\Theta(c^n)$	Time grows exponentially — increasing the problem size by 1 results in a c-fold increase in the time required; doubling the problem size *squares* the time required	Traveling salesman problem

The analysis of an algorithm is often accomplished by finding and solving a recurrence relation that describes the time required by the algorithm. The most commonly occurring families of recurrences in the analysis of algorithms are linear recurrences and divide-and-conquer recurrences. In the following subsection we describe the "method of operators" for solving linear recurrences; in the next subsection we describe how to transform divide-and-conquer recurrences into linear recurrences by substitution to obtain an asymptotic solution.

3.2.1 Linear Recurrences

A *linear recurrence with constant coefficients* has the form

$$c_0 a_n + c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} = f(n), \tag{3.1}$$

for some constant k, where each c_i is constant. To solve such a recurrence for a broad class of functions f (that is, to express a_n in closed form as a function of n) by the *method of operators*, we consider two basic operators on sequences: $\mathcal{S}$, which shifts the sequence left,

$$\mathcal{S}\langle a_0, a_1, a_2, \ldots\rangle = \langle a_1, a_2, a_3, \ldots\rangle,$$

and C, which, for any constant C, multiplies each term of the sequence by C:

$$C\langle a_0, a_1, a_2, \ldots\rangle = \langle Ca_0, Ca_1, Ca_2, \ldots\rangle.$$

Then, given operators A and B, we define the sum and product

$$\begin{aligned}(A+B)\langle a_0, a_1, a_2, \ldots\rangle &= A\langle a_0, a_1, a_2, \ldots\rangle + B\langle a_0, a_1, a_2, \ldots\rangle,\\ (AB)\langle a_0, a_1, a_2, \ldots\rangle &= A(B\langle a_0, a_1, a_2, \ldots\rangle).\end{aligned}$$

Thus, for example,

$$(\mathcal{S}^2 - 4)\langle a_0, a_1, a_2, \ldots\rangle = \langle a_2 - 4a_0, a_3 - 4a_1, a_4 - 4a_2, \ldots\rangle,$$

which we write more briefly as

$$(\mathcal{S}^2 - 4)\langle a_i\rangle = \langle a_{i+2} - 4a_i\rangle.$$

With the operator notation, we can rewrite Equation (3.1) as

$$P(\mathcal{S})\langle a_i \rangle = \langle f(i) \rangle,$$

where

$$P(\mathcal{S}) = c_0\mathcal{S}^k + c_1\mathcal{S}^{k-1} + c_2\mathcal{S}^{k-2} + \cdots + c_k$$

is a polynomial in $\mathcal{S}$.

Given a sequence $\langle a_i \rangle$, we say that the operator $P(\mathcal{S})$ *annihilates* $\langle a_i \rangle$ if $P(\mathcal{S})\langle a_i \rangle = \langle 0 \rangle$. For example, $\mathcal{S}^2 - 4$ annihilates any sequence of the form $\langle u2^i + v(-2)^i \rangle$, with constants u and v. In general,

The operator $\mathcal{S}^{k+1} - c$ annihilates $\langle c^i \times$ a polynomial in i of degree $k \rangle$.

The *product* of two annihilators annihilates the *sum* of the sequences annihilated by each of the operators, that is, if A annihilates $\langle a_i \rangle$ and B annihilates $\langle b_i \rangle$, then AB annihilates $\langle a_i + b_i \rangle$. Thus, determining the annihilator of a sequence is tantamount to determining the sequence; moreover, it is straightforward to determine the annihilator from a recurrence relation.

For example, consider the Fibonacci recurrence

$$F_0 = 0$$
$$F_1 = 1$$
$$F_{i+2} = F_{i+1} + F_i.$$

The last line of this definition can be rewritten as $F_{i+2} - F_{i+1} - F_i = 0$, which tells us that $\langle F_i \rangle$ is annihilated by the operator

$$\mathcal{S}^2 - \mathcal{S} - 1 = (\mathcal{S} - \phi)(\mathcal{S} + 1/\phi),$$

where $\phi = (1 + \sqrt{5})/2$. Thus we conclude that

$$F_i = u\phi^i + v(-\phi)^{-i}$$

for some constants u and v. We can now use the initial conditions $F_0 = 0$ and $F_1 = 1$ to determine u and v: These initial conditions mean that

$$u\phi^0 + v(-\phi)^{-0} = 0$$
$$u\phi^1 + v(-\phi)^{-1} = 1$$

and these linear equations have the solution

$$u = v = 1/\sqrt{5},$$

and hence

$$F_i = \phi^i/\sqrt{5} + (-\phi)^{-i}/\sqrt{5}.$$

In the case of the similar recurrence,

$$G_0 = 0$$
$$G_1 = 1$$
$$G_{i+2} = G_{i+1} + G_i + i,$$

TABLE 3.2 Rate of Growth of the Solution to the Recurrence $T(n) = g(n) + uT(n/v)$: The Divide-and-Conquer Recurrence Relations

$g(n)$	u, v	Growth Rate of $T(n)$
$\Theta(1)$	$u = 1$	$\Theta(\log n)$
	$u \neq 1$	$\Theta(n^{\log_v u})$
$\Theta(\log n)$	$u = 1$	$\Theta[(\log n)^2]$
	$u \neq 1$	$\Theta(n^{\log_v u})$
$\Theta(n)$	$u < v$	$\Theta(n)$
	$u = v$	$\Theta(n \log n)$
	$u > v$	$\Theta(n^{\log_v u})$
$\Theta(n^2)$	$u < v^2$	$\Theta(n^2)$
	$u = v^2$	$\Theta(n^2 \log n)$
	$u > v^2$	$\Theta(n^{\log_v u})$

u and v are positive constants, independent of n, and $v > 1$.

the last equation tells us that

$$(\mathcal{S}^2 - \mathcal{S} - 1)\langle G_i \rangle = \langle i \rangle,$$

so the annihilator for $\langle G_i \rangle$ is $(\mathcal{S}^2 - \mathcal{S} - 1)(\mathcal{S} - 1)^2$ since $(\mathcal{S} - 1)^2$ annihilates $\langle i \rangle$ (a polynomial of degree 1 in i) and hence the solution is

$$G_i = u\phi^i + v(-\phi)^{-i} + \text{(a polynomial of degree 1 in i)};$$

that is,

$$G_i = u\phi^i + v(-\phi)^{-i} + wi + z.$$

Again, we use the initial conditions to determine the constants u, v, w, and x.

In general, then, to solve the recurrence in Equation 3.1, we factor the annihilator

$$P(\mathcal{S}) = c_0\mathcal{S}^k + c_1\mathcal{S}^{k-1} + c_2\mathcal{S}^{k-2} + \cdots + c_k,$$

multiply it by the annihilator for $\langle f(i) \rangle$, write the form of the solution from this product (which is the annihilator for the sequence $\langle a_i \rangle$), and the use the initial conditions for the recurrence to determine the coefficients in the solution.

3.2.2 Divide-and-Conquer Recurrences

The divide-and-conquer paradigm of algorithm construction that we discuss in Section 4 leads naturally to divide-and-conquer recurrences of the type

$$T(n) = g(n) + uT(n/v),$$

for constants u and v, $v > 1$, and sufficient initial values to define the sequence $\langle T(0), T(1), T(2), \ldots \rangle$. The growth rates of $T(n)$ for various values of u and v are given in Table 3.2. The growth rates in this table are derived by transforming the divide-and-conquer recurrence into a linear recurrence for a subsequence of $\langle T(0), T(1), T(2), \ldots \rangle$.

To illustrate this method, we derive the penultimate line in Table 3.2. We want to solve

$$T(n) = n^2 + v^2 T(n/v).$$

So, we want to find a subsequence of $\langle T(0), T(1), T(2), \ldots\rangle$ that will be easy to handle. Let $n_k = v^k$; then,

$$T(n_k) = n_k^2 + v^2 T(n_k/v),$$

or

$$T(v^k) = v^{2k} + v^2 T(v^{k-1}).$$

Defining $t_k = T(v^k)$,

$$t_k = v^{2k} + v^2 t_{k-1}.$$

The annihilator for t_k is then $(\mathcal{S} - v^2)^2$ and thus

$$t_k = v^{2k}(ak + b),$$

for constants a and b. Expressing this in terms of $T(n)$,

$$T(n) \approx t_{\log_v n} = v^{2\log_v n}(a\log_v n + b) = an^2 \log_v n + bn^2,$$

or,

$$T(n) = \Theta(n^2 \log n).$$

3.3 Some Examples of the Analysis of Algorithms

In this section we introduce the basic ideas of analyzing algorithms by looking at some data structure problems that commonly occur in practice, problems relating to maintaining a collection of n objects and retrieving objects based on their relative size. For example, how can we determine the smallest of the elements? Or, more generally, how can we determine the kth largest of the elements? What is the running time of such algorithms in the worst case? Or, on average, if all $n!$ permutations of the input are equally likely? What if the set of items is dynamic — that is, the set changes through insertions and deletions — how efficiently can we keep track of, say, the largest element?

3.3.1 Sorting

The most demanding request that we can make of an array of n values `x[1]`, `x[2]`, ..., `x[n]` is that they be kept in perfect order so that `x[1]` $\leq$ `x[2]` $\leq \cdots \leq$ `x[n]`. The simplest way to put the values in order is to mimic what we might do by hand: take item after item and insert each one into the proper place among those items already inserted:

```
void insert (float x[], int i, float a) {
 // Insert a into x[1] ... x[i]
 // x[1] ... x[i-1] are sorted;  x[i] is unoccupied
 if (i == 1 || x[i-1] <= a)
  x[i] = a;
 else {
  x[i] = x[i-1];
  insert(x, i-1, a);
 }
}

void insertionSort (int n, float x[]) {
 // Sort  x[1] ... x[n]
```

```
  if (n > 1) {
    insertionSort(n-1, x);
    insert(x, n, x[n]);
  }
}
```

To determine the time required in the worst case to sort n elements with `insertionSort`, we let t_n be the time to sort n elements and derive and solve a recurrence relation for t_n. We have,

$$t_n \begin{cases} \Theta(1) & \text{if } n = 1, \\ t_{n-1} + s_{n-1} + \Theta(1) & \text{otherwise,} \end{cases}$$

where s_m is the time required to insert an element in place among m elements using `insert`. The value of s_m is also given by a recurrence relation:

$$s_m \begin{cases} \Theta(1) & \text{if } m = 1, \\ s_{m-1} + \Theta(1) & \text{otherwise.} \end{cases}$$

The annihilator for $\langle s_i \rangle$ is $(\mathcal{S}-1)^2$, so $s_m = \Theta(m)$. Thus, the annihilator for $\langle t_i \rangle$ is $(\mathcal{S}-1)^3$, so $t_n = \Theta(n^2)$. The analysis of the average behavior is nearly identical; only the constants hidden in the Θ-notation change.

We can design better sorting methods using the divide-and-conquer idea of the next section. These algorithms avoid $\Theta(n^2)$ worst-case behavior, working in time $\Theta(n \log n)$. We can also achieve time $\Theta(n \log n)$ using a clever way of viewing the array of elements to be sorted as a tree: consider `x[1]` as the root of the tree and, in general, `x[2*i]` is the root of the left subtree of `x[i]` and `x[2*i+1]` is the root of the right subtree of `x[i]`. If we further insist that parents be greater than or equal to children, we have a *heap*; Figure 3.1 shows a small example.

A heap can be used for sorting by observing that the largest element is at the root, that is, `x[1]`; thus, to put the largest element in place, we swap `x[1]` and `x[n]`. To continue, we must restore the heap property, which may now be violated at the root. Such restoration is accomplished by swapping `x[1]` with its larger child, if that child is larger than `x[1]`, and the continuing to swap it downward until either it reaches the bottom or a spot where it is greater or equal to its children. Because the tree-cum-array has height $\Theta(\log n)$, this restoration process takes time $\Theta(\log n)$. Now, with the heap in `x[1]` to `x[n-1]` and `x[n]` the largest value in the array, we can put the second largest element in place by swapping `x[1]` and `x[n-1]`; then we restore the heap property in `x[1]` to `x[n-2]` by propagating `x[1]` downward; this takes time $\Theta(\log(n-1))$. Continuing in this fashion, we find we can sort the entire array in time

$$\Theta(\log n + \log(n-1) + \cdots + \log 1) = \Theta(n \log n).$$

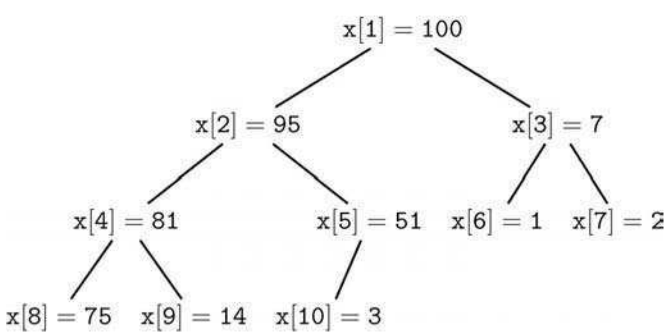


FIGURE 3.1 A heap — that is, an array, interpreted as a binary tree.

The initial creation of the heap from an unordered array is done by applying the restoration process successively to `x[n/2]`, `x[n/2-1]`, ..., `x[1]`, which takes time $\Theta(n)$.

Hence, we have the following $\Theta(n \log n)$ sorting algorithm:

```
void heapify (int n, float x[], int i) {
  // Repair heap property below x[i] in x[1] ... x[n]
  int largest = i;  // largest of x[i], x[2*i], x[2*i+1]
  if (2*i <= n && x[2*i] > x[i])
    largest = 2*i;
  if (2*i+1 <= n && x[2*i+1] > x[largest])
    largest = 2*i+1;
  if (largest != i) {
    // swap x[i] with larger child and repair heap below
    float t = x[largest]; x[largest] = x[i]; x[i] = t;
    heapify(n, x, largest);
  }
}

void makeheap (int n, float x[]) {
  // Make x[1] ... x[n] into a heap
  for (int i=n/2; i>0; i--)
    heapify(n, x, i);
}

void heapsort (int n, float x[]) {
  // Sort  x[1] ... x[n]
  float t;
  makeheap(n, x);
  for (int i=n; i>1; i--) {
    // put x[1] in place and repair heap
    t = x[1]; x[1] = x[i]; x[i] = t;
    heapify(i-1, x, 1);
  }
}
```

Can we find sorting algorithms that take less time than $\Theta(n \log n)$? The answer is no if we are restricted to sorting algorithms that derive their information from comparisons between the values of elements. The flow of control in such sorting algorithms can be viewed as binary trees in which there are $n!$ leaves, one for every possible sorted output arrangement. Because a binary tree with height h can have at most 2^h leaves, it follows that the height of a tree with $n!$ leaves must be at least $\log_2 n! = \Theta(n \log n)$. Because the height of this tree corresponds to the longest sequence of element comparisons possible in the flow of control, any such sorting algorithm must, in its worst case, use time proportional to $n \log n$.

3.3.2 Priority Queues

Aside from its application to sorting, the heap is an interesting data structure in its own right. In particular, heaps provide a simple way to implement a *priority queue*; a priority queue is an abstract data structure that keeps track of a dynamically changing set of values allowing the operations

`create`: Create an empty priority queue.
`insert`: Insert a new element into a priority queue.
`decrease`: Decrease an element in a priority queue.
`minimum`: Report the minimum element in a priority queue.

`deleteMinimum`: Delete the minimum element in a priority queue.
`delete`: Delete an element in a priority queue.
`merge`: Merge two priority queues.

A heap can implement a priority queue by altering the heap property to insist that parents are less than or equal to their children, so that that smallest value in the heap is at the root, that is, in the first array position. Creation of an empty heap requires just the allocation of an array, an $\Theta(1)$ operation; we assume that once created, the array containing the heap can be extended arbitrarily at the right end. Inserting a new element means putting that element in the $(n+1)$st location and "bubbling it up" by swapping it with its parent until it reaches either the root or a parent with a smaller value. Because a heap has logarithmic height, insertion to a heap of n elements thus requires worst-case time $O(\log n)$. Decreasing a value in a heap requires only a similar $O(\log n)$ "bubbling up." The minimum element of such a heap is always at the root, so reporting it takes $\Theta(1)$ time. Deleting the minimum is done by swapping the first and last array positions, bubbling the new root value downward until it reaches its proper location, and truncating the array to eliminate the last position. `Delete` is handled by decreasing the value so that it is the least in the heap and then applying the `deleteMinimum` operation; this takes a total of $O(\log n)$ time.

The `merge` operation, unfortunately, is not so economically accomplished; there is little choice but to create a new heap out of the two heaps in a manner similar to the `makeheap` function in heapsort. If there are a total of n elements in the two heaps to be merged, this re-creation will require time $O(n)$.

There are better data structures than a heap for implementing priority queues, however. In particular, the *Fibonacci heap* provides an implementation of priority queues in which the `delete` and `deleteMinimum` operations take $O(\log n)$ time and the remaining operations take $\Theta(1)$ time, *provided we consider the times required for a sequence of priority queue operations, rather than individual times.* That is, we must consider the cost of the individual operations *amortized over the sequence of operations*: Given a sequence of n priority queue operations, we will compute the total time $T(n)$ for all n operations. In doing this computation, however, we do not simply add the costs of the individual operations; rather, we subdivide the cost of each operation into two parts: the *immediate cost* of doing the operation and the *long-term savings* that result from doing the operation. The long-term savings represent costs *not* incurred by later operations as a result of the present operation. The immediate cost minus the long-term savings give the amortized cost of the operation.

It is easy to calculate the immediate cost (time required) of an operation, but how can we measure the long-term savings that result? We imagine that the data structure has associated with it a bank account; at any given moment, the bank account must have a non-negative balance. When we do an operation that will save future effort, we are making a deposit to the savings account; and when, later on, we derive the benefits of that earlier operation, we are making a withdrawal from the savings account. Let $\mathcal{B}(i)$ denote the balance in the account after the ith operation, $\mathcal{B}(0) = 0$. We define the amortized cost of the ith operation to be

$$\begin{aligned}\text{Amortized cost of } i\text{th operation} &= (\text{Immediate cost of } i\text{th operation}) + (\text{Change in bank account})\\ &= (\text{Immediate cost of } i\text{th operation}) + (\mathcal{B}(i) - \mathcal{B}(i-1)).\end{aligned}$$

Because the bank account $\mathcal{B}$ can go up or down as a result of the ith operation, the amortized cost may be less than or more than the immediate cost. By summing the previous equation, we get

$$\begin{aligned}\sum_{i=1}^{n}(\text{Amortized cost of } i\text{th operation}) &= \sum_{i=1}^{n}(\text{Immediate cost of } i\text{th operation}) + (\mathcal{B}(n) - \mathcal{B}(0))\\ &= (\text{Total cost of all } n \text{ operations}) + \mathcal{B}(n)\\ &\geq \text{Total cost of all } n \text{ operations}\\ &= T(n)\end{aligned}$$

because $\mathcal{B}(i)$ is non-negative. Thus defined, the sum of the amortized costs of the operations gives us an upper bound on the total time $T(n)$ for all n operations.

It is important to note that the function $\mathcal{B}(i)$ is not part of the data structure, but is just our way to measure how much time is used by the sequence of operations. As such, we can choose *any rules* for $\mathcal{B}$, provided $\mathcal{B}(0) = 0$ and $\mathcal{B}(i) \geq 0$ for $i \geq 1$. Then the sum of the amortized costs defined by

$$\text{Amortized cost of } i\text{th operation} = (\text{Immediate cost of } i\text{th operation}) + (\mathcal{B}(i) - \mathcal{B}(i-1))$$

bounds the overall cost of the operation of the data structure.

Now to apply this method to priority queues. A *Fibonacci heap* is a list of heap-ordered trees (not necessarily binary); because the trees are heap ordered, the minimum element must be one of the roots and we keep track of which root is the overall minimum. Some of the tree nodes are *marked*. We define

$$\begin{aligned}\mathcal{B}(i) = {} & (\text{Number of trees after the } i\text{th operation}) \\ & + 2 \times (\text{Number of marked nodes after the } i\text{th operation}).\end{aligned}$$

The clever rules by which nodes are marked and unmarked, and the intricate algorithms that manipulate the set of trees, are too complex to present here in their complete form, so we just briefly describe the simpler operations and show the calculation of their amortized costs:

`Create`: To create an empty Fibonacci heap we create an empty list of heap-ordered trees. The immediate cost is $\Theta(1)$; because the numbers of trees and marked nodes are zero before and after this operation, $\mathcal{B}(i) - \mathcal{B}(i-1)$ is zero and the amortized time is $\Theta(1)$.

`Insert`: To insert a new element into a Fibonacci heap we add a new one-element tree to the list of trees constituting the heap and update the record of what root is the overall minimum. The immediate cost is $\Theta(1)$. $\mathcal{B}(i) - \mathcal{B}(i-1)$ is also 1 because the number of trees has increased by 1, while the number of marked nodes is unchanged. The amortized time is thus $\Theta(1)$.

`Decrease`: Decreasing an element in a Fibonacci heap is done by cutting the link to its parent, if any, adding the item as a root in the list of trees, and decreasing its value. Furthermore, the marked parent of a cut element is itself cut, propagating upward in the tree. Cut nodes become unmarked, and the unmarked parent of a cut element becomes marked. The immediate cost of this operation is $\Theta(c)$, where c is the number of cut nodes. If there were t trees and m marked elements before this operation, the value of $\mathcal{B}$ before the operation was $t + 2m$. After the operation, the value of $\mathcal{B}$ is $(t+c)+2(m-c+2)$, so $\mathcal{B}(i) - \mathcal{B}(i-1) = 4-c$. The amortized time is thus $\Theta(c)+4-c = \Theta(1)$ *by changing the definition of* $\mathcal{B}$ *by a multiplicative constant large enough to dominate the constant hidden in* $\Theta(c)$.

`Minimum`: Reporting the minimum element in a Fibonacci heap takes time $\Theta(1)$ and does not change the numbers of trees and marked nodes; the amortized time is thus $\Theta(1)$.

`DeleteMinimum`: Deleting the minimum element in a Fibonacci heap is done by deleting that tree root, making its children roots in the list of trees. Then, the list of tree roots is "consolidated" in a complicated $O(\log n)$ operation that we do not describe. The result takes amortized time $O(\log n)$.

`Delete`: Deleting an element in a Fibonacci heap is done by decreasing its value to $-\infty$ and then doing a `deleteMinimum`. The amortized cost is the sum of the amortized cost of the two operations, $O(\log n)$.

`Merge`: Merging two Fibonacci heaps is done by concatenating their lists of trees and updating the record of which root is the minimum. The amortized time is thus $\Theta(1)$.

Notice that the amortized cost of each operation is $\Theta(1)$ except `deleteMinimum` and `delete`, both of which are $O(\log n)$.

3.4 Divide-and-Conquer Algorithms

One approach to the design of algorithms is to decompose a problem into subproblems that resemble the original problem, but on a reduced scale. Suppose, for example, that we want to compute x^n. We reason that the value we want can be computed from $x^{\lfloor n/2 \rfloor}$ because

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ (x^{\lfloor n/2 \rfloor})2 & \text{if } n \text{ is even,} \\ x \times (x^{\lfloor n/2 \rfloor})2 & \text{if } n \text{ is odd.} \end{cases}$$

This recursive definition can be translated directly into

```
 1  int power (float x, int n) {
 2   // Compute the n-th power of x
 3    if (n == 0)
 4      return 1;
 5    else {
 6      int t = power(x, floor(n/2));
 7      if ((n % 2) == 0)
 8        return t*t;
 9      else
10        return x*t*t;
11    }
12  }
```

To analyze the time required by this algorithm, we notice that the time will be proportional to the number of multiplication operations performed in lines 8 and 10, so the divide-and-conquer recurrence

$$T(n) = 2 + T(\lfloor n/2 \rfloor),$$

with $T(0) = 0$, describes the rate of growth of the time required by this algorithm. By considering the subsequence $n_k = 2^k$, we find, using the methods of the previous section, that $T(n) = \Theta(\log n)$. Thus, the above algorithm is considerably more efficient than the more obvious

```
int power (int k, int n) {
 // Compute the n-th power of k
  int product = 1;
  for (int i = 1; i <= n; i++)
    // at this point power is k*k*k*...*k (i times)
    product = product * k;
  return product;
}
```

which requires time $\Theta(n)$.

An extremely well-known instance of divide-and-conquer algorithms is *binary search* of an ordered array of n elements for a given element; we "probe" the middle element of the array, continuing in either the lower or upper segment of the array, depending on the outcome of the probe:

```
int binarySearch (int x, int w[], int low, int high) {
  // Search for x among sorted array w[low..high]. The integer returned
  // is either the location of x in w, or the location where x belongs.
  if (low > high) // Not found
    return low;
```

```
  else {
    int middle := (low+high)/2;
    if (w[middle] < x)
      return binarySearch(x, w, middle+1, high);
    else if (w[middle] == x)
      return middle;
    else
      return binarySearch(x, w, low, middle-1);
  }
}
```

The analysis of binary search in an array of n elements is based on counting the number of probes used in the search, because all remaining work is proportional to the number of probes. But, the number of probes needed is described by the divide-and-conquer recurrence

$$T(n) = 1 + T(n/2),$$

with $T(0) = 0$, $T(1) = 1$. We find from Table 3.2 (the top line) that $T(n) = \Theta(\log n)$. Hence, binary search is much more efficient than a simple linear scan of the array.

To multiply two very large integers x and y, assume that x has exactly $l \geq 2$ digits and y has at most l digits. Let $x_0, x_1, x_2, \ldots, x_{l-1}$ be the digits of x and let $y_0, y_1, \ldots, y_{l-1}$ be the digits of y (some of the significant digits at the end of y may be zeros, if y is shorter than x), so that

$$x = x_0 + 10x_1 + 10^2 x_2 + \cdots + 10^{l-1} x_{l-1},$$

and

$$y = y_0 + 10y_1 + 10^2 y_2 + \cdots + 10^{l-1} y_{l-1},$$

We apply the divide-and-conquer idea to multiplication by chopping x into two pieces — the leftmost n digits and the remaining digits:

$$x = x_{\text{left}} + 10^n x_{\text{right}},$$

where $n = l/2$. Similarly, chop y into two corresponding pieces:

$$y = y_{\text{left}} + 10^n y_{\text{right}},$$

because y has at most the number of digits that x does, y_{right} might be 0. The product $x \times y$ can be now written

$$\begin{aligned} x \times y &= (x_{\text{left}} + 10^n x_{\text{right}}) \times (y_{\text{left}} + 10^n y_{\text{right}}), \\ &= x_{\text{left}} \times y_{\text{left}} \\ &\quad + 10^n (x_{\text{right}} \times y_{\text{left}} + x_{\text{left}} \times y_{\text{right}}) \\ &\quad + 10^{2n} x_{\text{right}} \times y_{\text{right}}. \end{aligned}$$

If $T(n)$ is the time to multiply two n-digit numbers with this method, then

$$T(n) = kn + 4T(n/2);$$

the kn part is the time to chop up x and y and to do the needed additions and shifts; each of these tasks involves n-digit numbers and hence $\Theta(n)$ time. The $4T(n/2)$ part is the time to form the four needed subproducts, each of which is a product of about $n/2$ digits.

The line for $g(n) = \Theta(n)$, $u = 4 > v = 2$ in Table 3.2 tells us that $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$, so the divide-and-conquer algorithm is no more efficient than the elementary-school method of multiplication. However, we can be more economical in our formation of subproducts:

$$\begin{aligned} x \times y &= \left(x_{\text{left}} + 10^n x_{\text{right}}\right) \times \left(y_{\text{left}} + 10^n y_{\text{right}}\right), \\ &= B + 10^n C + 10^{2n} A, \end{aligned}$$

where

$$\begin{aligned} A &= x_{\text{right}} \times y_{\text{right}} \\ B &= x_{\text{left}} \times y_{\text{left}} \\ C &= (x_{\text{left}} + x_{\text{right}}) \times (y_{\text{left}} + y_{\text{right}}) - A - B. \end{aligned}$$

The recurrence for the time required changes to

$$T(n) = kn + 3T(n/2).$$

The kn part is the time to do the two additions that form $x \times y$ from A, B, and C and the two additions and the two subtractions in the formula for C; each of these six additions/subtractions involves n-digit numbers. The $3T(n/2)$ part is the time to (recursively) form the three needed products, each of which is a product of about $n/2$ digits. The line for $g(n) = \Theta(n)$, $u = 3 > v = 2$ in Table 3.2 now tells us that

$$T(n) = \Theta\left(n^{\log_2 3}\right).$$

Now,

$$\log_2 3 = \frac{\log_{10} 3}{\log_{10} 2} \approx 1.5849625\cdots,$$

which means that this divide-and-conquer multiplication technique will be faster than the straightforward $\Theta(n^2)$ method for large numbers of digits.

Sorting a sequence of n values efficiently can be done using the divide-and-conquer idea. Split the n values arbitrarily into two piles of $n/2$ values each, sort each of the piles separately, and then merge the two piles into a single sorted pile. This sorting technique, pictured in Figure 3.2, is called *merge sort*. Let $T(n)$ be the time required by merge sort for sorting n values. The time needed to do the merging is proportional to the number of elements being merged, so that

$$T(n) = cn + 2T(n/2),$$

because we must sort the two halves (time $T(n/2)$ each) and then merge (time proportional to n). We see by Table 3.2 that the growth rate of $T(n)$ is $\Theta(n \log n)$, since $u = v = 2$ and $g(n) = \Theta(n)$.

3.5 Dynamic Programming

In the design of algorithms to solve optimization problems, we need to make the optimal (lowest cost, highest value, shortest distance, etc.) choice from among a large number of alternative solutions. *Dynamic programming* is an organized way to find an optimal solution by systematically exploring all possibilities without unnecessary repetition. Often, dynamic programming leads to efficient, polynomial-time algorithms for problems that appear to require searching through exponentially many possibilities.

Like the divide-and-conquer method, dynamic programming is based on the observation that many optimization problems can be solved by solving similar subproblems and the composing the solutions of those subproblems into a solution for the original problem. In addition, the problem is viewed as

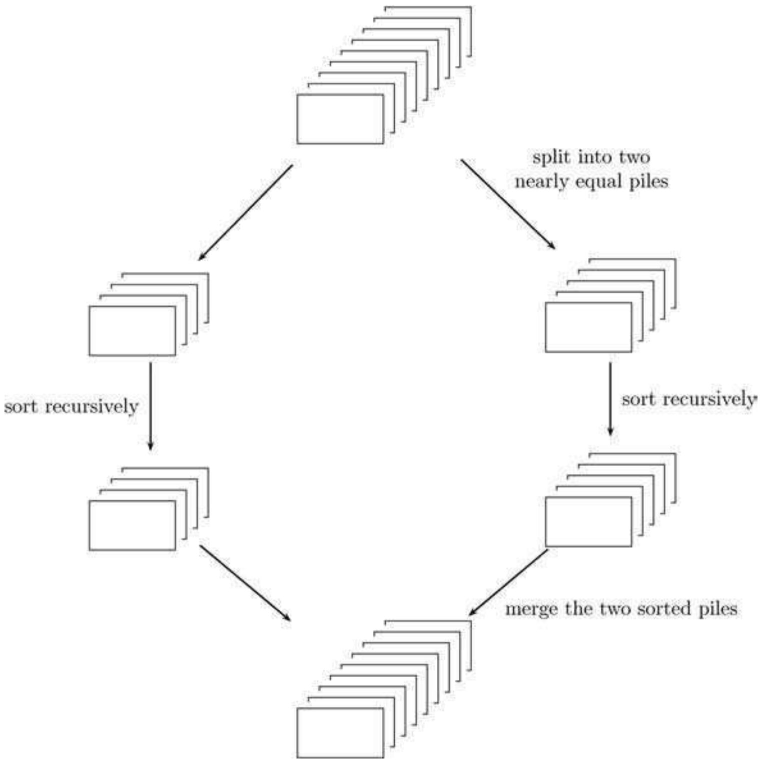


FIGURE 3.2 Schematic description of merge sort.

a sequence of decisions, each decision leading to different subproblems; if a wrong decision is made, a suboptimal solution results, so all possible decisions need to be accounted for.

As an example of dynamic programming, consider the problem of constructing an optimal search pattern for probing an ordered sequence of elements. The problem is similar to searching an array. In the previous section we described binary search, in which an interval in an array is repeatedly bisected until the search ends. Now, however, suppose we know the frequencies with which the search will seek various elements (both in the sequence and missing from it). For example, if we know that the last few elements in the sequence are frequently sought — binary search does not make use of this information — it might be more efficient to begin the search at the right end of the array, not in the middle. Specifically, we are given an ordered sequence $x_1 < x_2 < \cdots < x_n$ and associated frequencies of access $\beta_1, \beta_2, \ldots, \beta_n$, respectively; furthermore, we are given $\alpha_0, \alpha_1, \ldots, \alpha_n$ where α_i is the frequency with which the search will fail because the object sought, z, was missing from the sequence, $x_i < z < x_{i+1}$ (with the obvious meaning when $i = 0$ or $i = n$). What is the optimal order to search for an unknown element z? In fact, how should we describe the optimal search order?

We express a search order as a *binary search tree*, a diagram showing the sequence of probes made in every possible search. We place at the root of the tree the sequence element at which the first probe is made, for example, x_i; the left subtree of x_i is constructed recursively for the probes made when $z < x_i$, and the right subtree of x_i is constructed recursively for the probes made when $z > x_i$. We label each item in the tree with the frequency that the search ends at that item. Figure 3.3 shows a simple example. The search of sequence $x_1 < x_2 < x_3 < x_4 < x_5$ according the tree of Figure 3.3 is done by comparing the unknown element z with x_4 (the root); if $z = x_4$, the search ends. If $z < x_2$, z is compared with x_2 (the root of the left subtree); if $z = x_2$, the search ends. Otherwise, if $z < x_2$, z is compared with x_1 (the root of the left subtree of x_2); if $z = x_1$, the search ends. Otherwise, if $z < x_1$, the search ends unsuccessfully at the leaf labeled α_0. Other results of comparisons lead along other paths in the tree from the root downward. By its

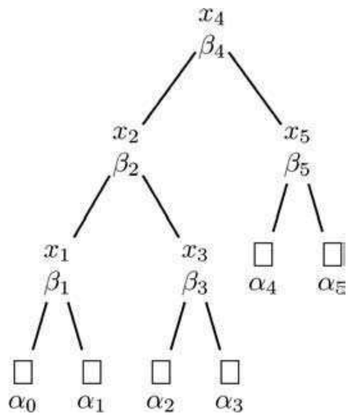


FIGURE 3.3 A binary search tree.

nature, a binary search tree is *lexicographic* in that for all nodes in the tree, the elements in the left subtree of the node are smaller and the elements in the right subtree of the node are larger than the node.

Because we are to find an optimal search pattern (tree), we want the cost of searching to be minimized. The cost of searching is measured by the *weighted path length* of the tree:

$$\sum_{i=1}^{n} \beta_i \times [1 + \text{level}(\beta_i)] + \sum_{i=0}^{n} \alpha_i \times \text{level}(\alpha_i),$$

defined formally as

$$W(\square) = 0,$$

$$W\left(T = {\overset{\bullet}{\diagup\diagdown} \atop T_l \ T_r}\right) = W(T_l) + W(T_r) + \sum \alpha_i + \sum \beta_i,$$

where the summations $\sum \alpha_i$ and $\sum \beta_i$ are over all α_i and β_i in T. Because there are exponentially many possible binary trees, finding the one with minimum weighted path length could, if done naïvely, take exponentially long.

The key observation we make is that a *principle of optimality* holds for the cost of binary search trees: subtrees of an optimal search tree must themselves be optimal. This observation means, for example, that if the tree shown in Figure 3.3 is optimal, then its left subtree must be the optimal tree for the problem of searching the sequence $x_1 < x_2 < x_3$ with frequencies $\beta_1, \beta_2, \beta_3$ and $\alpha_0, \alpha_1, \alpha_2, \alpha_3$. (If a subtree in Figure 3.3 were *not* optimal, we could replace it with a better one, reducing the weighted path length of the entire tree because of the recursive definition of weighted path length.) In general terms, the principle of optimality states that subsolutions of an optimal solution must themselves be optimal.

The optimality principle, together with the recursive definition of weighted path length, means that we can express the construction of an optimal tree recursively. Let $C_{i,j}$, $0 \le i \le j \le n$, be the cost of an optimal tree over $x_{i+1} < x_{i+2} < \cdots < x_j$ with the associated frequencies $\beta_{i+1}, \beta_{i+2}, \ldots, \beta_j$ and $\alpha_i, \alpha_{i+1}, \ldots, \alpha_j$. Then,

$$C_{i,i} = 0,$$
$$C_{i,j} = \min_{i<k\le j}(C_{i,k-1} + C_{k,j}) + W_{i,j},$$

where

$$W_{i,i} = \alpha_i,$$

$$W_{i,j} = W_{i,j-1} + \beta_j + \alpha_j.$$

These two recurrence relations can be implemented directly as recursive functions to compute $C_{0,n}$, the cost of the optimal tree, leading to the following two functions:

```
1  int W (int i, int j) {
2    if (i == j)
3      return alpha[j];
4    else
5      return W(i,j-1) + beta[j] + alpha[j];
6  }
7
8  int C (int i, int j) {
9    if (i == j)
10     return 0;
11   else {
12     int minCost = MAXINT;
13     int cost;
14     for (int k = i+1; k <= j; k++) {
15       cost = C(i,k-1) + C(k,j) + W(i,j);
16       if (cost < minCost)
17         minCost = cost;
18     }
19     return minCost;
20   }
21 }
```

These two functions correctly compute the cost of an optimal tree; the tree itself can be obtained by storing the values of `k` when `cost < minCost` in line 16.

However, the above functions are unnecessarily time consuming (requiring exponential time) because the same subproblems are solved repeatedly. For example, each call `W(i,j)` uses time $\Theta($`j` $-$ `i`$)$ and such calls are made repeatedly for the same values of `i` and `j`. We can make the process more efficient by caching the values of `W(i,j)` in an array as they are computed and using the cached values when possible:

```
int W[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    W[i][j] = MAXINT;

int W (int i, int j) {
  if (W[i][j] = MAXINT)
    if (i == j)
       W[i][j] = alpha[j];
    else
      W[i][j] =  W(i,j-1) + beta[j] + alpha[j];
  return W[i][j];
}
```

In the same way, we should cache the values of `C(i,j)` in an array as they are computed:

```
int C[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    C[i][j] = MAXINT;

```

```
int C (int i, int j) {
  if (C[i][j] == MAXINT)
    if (i == j)
       C[i][j] = 0;
    else {
      int minCost = MAXINT;
      int cost;
      for (int k = i+1; k <= j; k++) {
        cost = C(i,k-1) + C(k,j) + W(i,j);
        if (cost < minCost)
          minCost = cost;
      }
      C[i][j] = minCost;
    }
  return C[i][j];
}
```

The idea of caching the solutions to subproblems is crucial to making the algorithm efficient. In this case, the resulting computation requires time $\Theta(n^3)$; this is surprisingly efficient, considering that an optimal tree is being found from among exponentially many possible trees.

By studying the pattern in which the arrays `C` and `W` are filled in, we see that the main diagonal `C[i][i]` is filled in first, then the first upper super-diagonal `C[i][i+1]`, then the second upper super-diagonal `C[i][i+2]`, and so on until the upper-right corner of the array is reached. Rewriting the code to do this directly, and adding an array `R[][]` to keep track of the roots of subtrees, we obtain:

```
int W[n][n];
int R[n][n];
int C[n][n];

// Fill in main diagonal
for (int i = 0; i < n; i++) {
  W[i][i] = alpha[i];
  R[i][i] = 0;
  C[i][i] = 0;
}

int minCost, cost;
for (int d = 1; d < n; d++)
  // Fill in d-th upper super-diagonal
  for (i = 0; i < n-d; i++) {
    W[i][i+d] = W[i][i+d-1] + beta[i+d] + alpha[i+d];
    R[i][i+d] = i+1;
    C[i][i+d] = C[i][i] + C[i+1][i+d] + W[i][i+d];
    for (int k = i+2; k <= i+d; k++) {
      cost = C[i][k-1] + C[k][i+d] + W[i][i+d];
      if (cost < C[i][i+d]) {
        R[i][i+d] = k;
        C[i][i+d] = cost;
      }
    }
  }
```

which more clearly shows the $\Theta(n^3)$ behavior.

As a second example of dynamic programming, consider the *traveling salesman problem* in which a salesman must visit n cities, returning to his starting point, and is required to minimize the cost of the trip. The cost of going from city i to city j is $C_{i,j}$. To use dynamic programming we must specify an optimal tour in a recursive framework, with subproblems resembling the overall problem. Thus we define

$$T(i; j_1, j_2, \ldots, j_k) = \begin{cases} \text{cost of an optimal tour from city } i \text{ to city} \\ \text{1 that goes through each of the cities } j_1, \\ j_2, \ldots, j_k \text{ exactly once, in any order, and} \\ \text{through no other cities.} \end{cases}$$

The principle of optimality tells us that

$$T(i; j_1, j_2, \ldots, j_k) = \min_{1 \le m \le k} \{C_{i,j_m} + T(j_m; j_1, j_2, \ldots, j_{m-1}, j_{m+1}, \ldots, j_k)\},$$

where, by definition,

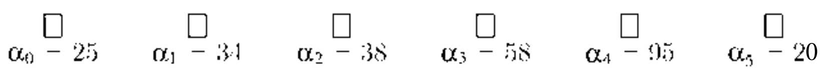
$$T(i; j) = C_{i,j} + C_{j,1}.$$

We can write a function `T` that directly implements the above recursive definition, but as in the optimal search tree problem, many subproblems would be solved repeatedly, leading to an algorithm requiring time $\Theta(n!)$. By caching the values $T(i; j_1, j_2, \ldots, j_k)$, we reduce the time required to $\Theta(n^2 2^n)$, still exponential, but considerably less than without caching.

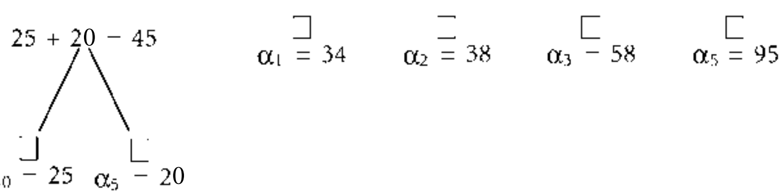
3.6 Greedy Heuristics

Optimization problems always have an objective function to be minimized or maximized, but it is not often clear what steps to take to reach the optimum value. For example, in the optimum binary search tree problem of the previous section, we used dynamic programming to systematically examine all possible trees. But perhaps there is a simple rule that leads directly to the best tree; say, by choosing the largest β_i to be the root and then continuing recursively. Such an approach would be less time-consuming than the $\Theta(n^3)$ algorithm we gave, but it does not necessarily give an optimum tree (if we follow the rule of choosing the largest β_i to be the root, we get trees that are no better, on the average, than a randomly chosen trees). The problem with such an approach is that it makes decisions that are *locally optimum*, although perhaps not *globally optimum*. But such a "greedy" sequence of locally optimum choices does lead to a globally optimum solution in some circumstances.

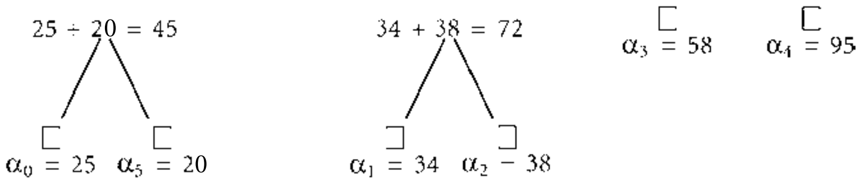
Suppose, for example, $\beta_i = 0$ for $1 \le i \le n$, and we remove the lexicographic requirement of the tree; the resulting problem is the determination of an optimal prefix code for $n + 1$ letters with frequencies $\alpha_0, \alpha_1, \ldots, \alpha_n$. Because we have removed the lexicographic restriction, the dynamic programming solution of the previous section no longer works, but the following simple greedy strategy yields an optimum tree: repeatedly combine the two lowest-frequency items as the left and right subtrees of a newly created item whose frequency is the sum of the two frequencies combined. Here is an example of this construction; we start with five leaves with weights



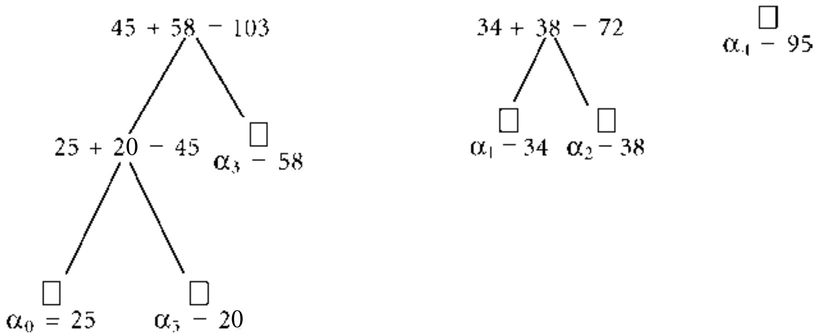
First, combine leaves $\alpha_0 = 25$ and $\alpha_5 = 20$ into a subtree of frequency $25 + 20 = 45$:



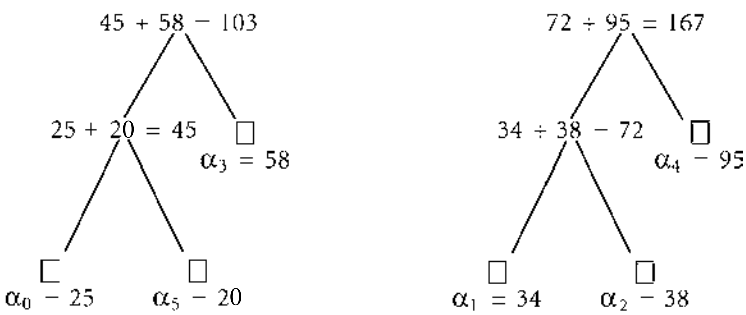
Then combine leaves $\alpha_1 = 34$ and $\alpha_2 = 38$ into a subtree of frequency $34 + 38 = 72$:



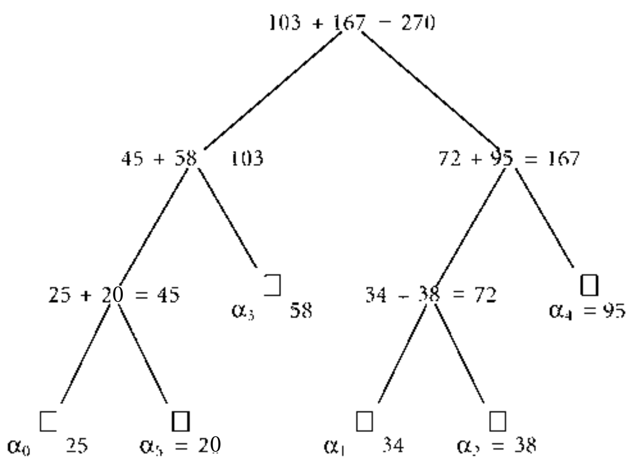
Next, combine the subtree of frequency $\alpha_0 + \alpha_5 = 45$ with $\alpha_3 = 58$:



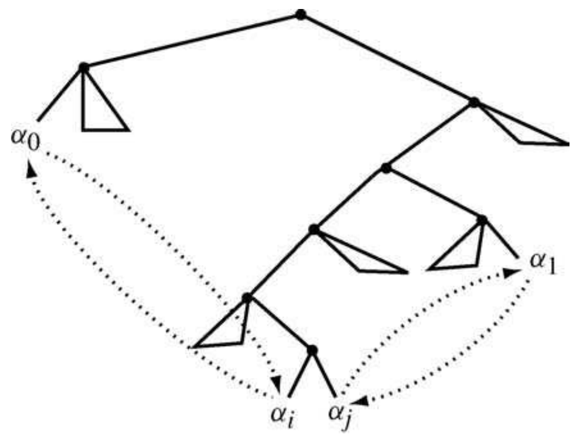
Then combine the subtree of frequency $\alpha_1 + \alpha_2 = 72$ with $\alpha_4 = 95$:



Finally, combine the only two remaining subtrees:



How do we know that the above-outlined process leads to an optimum tree? The key to proving that the tree is optimum is to assume, by way of contradiction, that it is not optimum. In this case, the greedy strategy must have erred in one of its choices, so let's look at the *first* error this strategy made. Because all previous greedy choices were not errors, and hence lead to an optimum tree, we can assume that we have a sequence of frequencies $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that the first greedy choice is erroneous — without loss of generality assume that α_0 and α_1 are two smallest frequencies, those combined erroneously by the greedy strategy. For this combination to be erroneous, there must be no optimum tree in which these two leaves are siblings, so consider an optimum tree, the locations of α_0 and α_1, and the location of the two deepest leaves in the tree, α_i and α_j:



By interchanging the positions of α_0 and α_i and α_1 and α_j (as shown), we obtain a tree in which α_0 and α_1 are siblings. Because α_0 and α_1 are the two lowest frequencies (because they were the greedy algorithm's choice) $\alpha_0 \leq \alpha_i$ and $\alpha_1 \leq \alpha_j$, the weighted path length of the modified tree is no larger than before the modification since $\text{level}(\alpha_0) \geq \text{level}(\alpha_i)$, $\text{level}(\alpha_1) \geq \text{level}(\alpha_j)$ and, hence,

$$\text{level}(\alpha_i) \times \alpha_0 + \text{level}(\alpha_j) \times \alpha_1 \leq \text{level}(\alpha_0) \times \alpha_0 + \text{level}(\alpha_1) \times \alpha_1.$$

In other words, the first so-called mistake of the greedy algorithm was in fact not a mistake because there is an optimum tree in which α_0 and α_1 are siblings. Thus we conclude that the greedy algorithm never makes a first mistake — that is, it never makes a mistake at all!

The greedy algorithm above is called *Huffman's algorithm*. If the subtrees are kept on a priority queue by cumulative frequency, the algorithm needs to insert the $n + 1$ leaf frequencies onto the queue, and then repeatedly remove the two least elements on the queue, unite those to elements into a single subtree, and put that subtree back on the queue. This process continues until the queue contains a single item, the optimum tree. Reasonable implementations of priority queues will yield $O(n \log n)$ implementations of Huffman's greedy algorithm.

The idea of making greedy choices, facilitated with a priority queue, works to find optimum solutions to other problems too. For example, a spanning tree of a weighted, connected, undirected graph $G = (V, E)$ is a subset of $|V| - 1$ edges from E connecting all the vertices in G; a spanning tree is minimum if the sum of the weights of its edges is as small as possible. *Prim's algorithm* uses a sequence of greedy choices to determine a minimum spanning tree: start with an arbitrary vertex $v \in V$ as the spanning-tree-to-be. Then, repeatedly add the cheapest edge connecting the spanning-tree-to-be to a vertex not yet in it. If the vertices not yet in the tree are stored in a priority queue implemented by a Fibonacci heap, the total time required by Prim's algorithm will be $O(|E| + |V| \log |V|)$. But why does the sequence of greedy choices lead to a minimum spanning tree?

Suppose Prim's algorithm does *not* result in a minimum spanning tree. As we did with Huffman's algorithm, we ask what the state of affairs must be when Prim's algorithm makes its first mistake; we will see that the assumption of a first mistake leads to a contradiction, thus proving the correctness of Prim's algorithm. Let the edges added to the spanning tree be, in the order added, $e_1, e_2, e_3, \ldots$, and let e_i be the first mistake. In other words, there is a minimum spanning tree T_{min} containing $e_1, e_2, \ldots, e_{i-1}$, but no minimum spanning tree contains $e_1, e_2, \ldots, e_i$. Imagine what happens if we add the edge e_i to T_{min}: because T_{min} is a spanning tree, the addition of e_i causes a cycle containing e_i. Let e_{max} be the highest-cost edge on that cycle. Because Prim's algorithm makes a greedy choice — that is, chooses the lowest cost available edge — the cost of e_{max} is at least that of e_i, so the cost of the spanning $T_{\text{min}} - \{e_{\text{max}}\} \cup \{e_i\}$ is at most that of T_{min}; in other words, $T_{\text{min}} - \{e_{\text{max}}\} \cup \{e_i\}$ is also a minimum spanning tree, contradicting our assumption that the choice of e_i is the first mistake. Therefore, the spanning tree constructed by Prim's algorithm must be a minimum spanning tree.

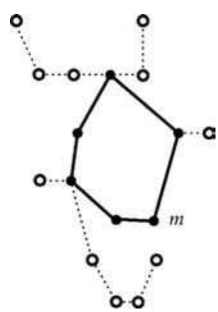
We can apply the greedy heuristic to many optimization problems, and even if the results are not optimal, they are often quite good. For example, in the n-city traveling salesman problem, we can get near-optimal tours in time $O(n^2)$ when the intercity costs are symmetric ($C_{i,j} = C_{j,i}$ for all i and j) and satisfy the triangle inequality ($C_{i,j} \leq C_{i,k} + C_{k,j}$ for all i, j, and k). The *closest insertion algorithm* starts with a "tour" consisting of a single, arbitrarily chosen city, and successively inserts the remaining cities to the tour, making a greedy choice about which city to insert next and where to insert it: the city chosen for insertion is the city not on the tour but closest to a city on the tour; the chosen city is inserted adjacent to the city on the tour to which it is closest.

Given an $n \times n$ symmetric distance matrix C that satisfies the triangle inequality, let I_n be the tour of length $|I_n|$ produced by the closest insertion heuristic and let O_n be an optimal tour of length $|O_n|$. Then,

$$\frac{|I_n|}{|O_n|} < 2.$$

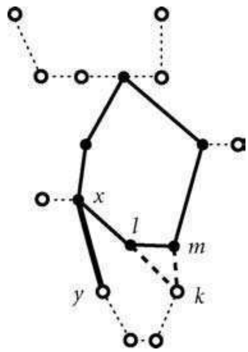
This bound is proved by an incremental form of the optimality proofs for greedy heuristics we saw seen above: we ask not where the first error is, but by how much we are in error at each greedy insertion to the tour; we establish a correspondence between edges of the optimal tour and cities inserted on the closest insertion tour. We show that at each insertion of a new city to the closest insertion tour, the cost of that insertion is at most twice the cost of corresponding edge of the optimal tour.

To establish the correspondence, imagine the closest insertion algorithm keeping track not only of the current tour, but also of a spider-like configuration including the edges of the current tour (the body of the spider) and pieces of the optimal tour (the legs of the spider). We show the current tour in solid lines and the pieces of optimal tour as dotted lines:



Initially, the spider consists of the arbitrarily chosen city with which the closest insertion tour begins and the legs of the spider consist of all the edges of the optimal tour *except* for one edge eliminated arbitrarily. As each city is inserted into the closest insertion tour, the algorithm will delete from the spider-like configuration one of the dotted edges from the optimal tour. When city k is inserted between cities l and m,

the edge deleted is the one attaching spider to the leg containing the city inserted (from city x to city y), shown here in bold:



Now,

$$C_{k,m} \leq C_{x,y}$$

because of the greedy choice to add city k to the tour and not city y. By the triangle inequality,

$$C_{l,k} \leq C_{l,m} + C_{m,k},$$

and by symmetry, we can combine these two inequalities to get

$$C_{l,k} \leq C_{l,m} + C_{x,y}.$$

Adding this last inequality to the first one above,

$$C_{l,k} + C_{k,m} \leq C_{l,m} + 2C_{x,y},$$

that is,

$$C_{l,k} + C_{k,m} - C_{l,m} \leq 2C_{x,y}.$$

Thus, adding city k between cities l and m adds no more to I_n than $2C_{x,y}$. Summing these incremental, amounts over the cost of the entire algorithm tells us that

$$I_n \leq 2O_n,$$

as we claimed.

References

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, McGraw-Hill, New York, 2nd ed., 2001.

Greene, D. H. and D. E. Knuth, *Mathematics for the Analysis of Algorithms*, 3rd ed., Birkhäuser, Boston, 1990.

Knuth, D. E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms,* Addison-Wesley, Reading, MA, 3rd ed., 1997.

Knuth, D. E., *The Art of Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley, Reading, MA, 2nd ed., 1998.

Lueker, G. S., "Some techniques for solving recurrences," *Computing Surveys*, **12**, 419–436, 1980.

Reingold, E. M. and W. J. Hansen, *Data Structures in Pascal*, Little, Brown and Company, Boston, 1986.

Reingold, E. M., J. Nievergelt, and N. Deo, *Combinatorial Algorithms*: *Theory and Practice*, Prentice Hall, Englewood Cliffs, NJ, 1977.

4

Data Structures

Roberto Tamassia
Brown University
Bryan M. Cantrill
Sun Microsystems, Inc.

4.1 Introduction

The study of data structures — that is, methods for organizing data that are suitable for computer processing — is one of the classic topics of computer science. At the hardware level, a computer views storage devices such as internal memory and disk as holders of elementary data units (bytes), each accessible through its address (an integer). When writing programs, instead of manipulating the data at the byte level, it is convenient to organize them into higher-level entities called *data structures*.

4.1.1 Containers, Elements, and Positions or Locators

Most data structures can be viewed as **containers** that store a collection of objects of a given type, called the *elements* of the container. Often, a total order is defined among the elements (e.g., alphabetically ordered names, points in the plane ordered by x-coordinate). Following the approach of Goodrich and Tamassia [2001], we assume that the elements of a container can be accessed by means of variables called **positions** or **locators**. When an object is inserted into the container, a position or locator is returned, which can be later used to access or delete the object. A position represents a "place" where an element is stored, Examples of positions are array cells and list nodes. A locator "tracks" the position of an element in the data structure as it changes over time. A locator is typically implemented with an object that stores a pointer to a position.

A data structure has an associated repertory of operations, classified into *queries*, which retrieve information on the data structure (e.g., return the number of elements, or test the presence of a given element), and *updates*, which modify the data structure (e.g., insertion and deletion of elements). The performance

of a data structure is characterized by the space requirement and the time complexity of the operations in its repertory. The *amortized* time complexity of an operation is the average time over a suitably defined sequence of operations.

However, efficiency is not the only quality measure of a data structure. Simplicity and ease of implementation should be taken into account when choosing a data structure for solving a practical problem.

4.1.2 Abstract Data Types

Data structures are concrete implementations of **abstract data types** (ADTs). A *data type* is a collection of objects. A data type can be mathematically specified (e.g., real number, directed graph) or concretely specified within a programming language (e.g., **int** in C, **set** in Pascal). An ADT is a mathematically specified data type equipped with operations that can be performed on the objects. Object-oriented programming languages, such as C++, provide support for expressing ADTs by means of *classes*. ADTs specify the data stored and the operations to be performed on them.

4.1.3 Main Issues in the Study of Data Structures

The following issues are of foremost importance in the study of data structures.

4.1.3.1 Static vs. Dynamic

A *static* data structure supports only queries, whereas a *dynamic* data structure also supports updates. A *dynamic* data structure is often more complicated than its static counterpart supporting the same repertory of queries. A *persistent* data structure (see, e.g., Driscoll et al. [1989]) is a dynamic data structure that supports operations on past versions. There are many problems for which no efficient dynamic data structures are known.

4.1.3.2 Implicit vs. Explicit

Two fundamental data organization mechanisms are used in data structures. In an *explicit* data structure, pointers (i.e., memory addresses) are used to link the elements and access them (e.g., a singly linked list, where each element has a pointer to the next one). In an *implicit* data structure (see, e.g., [Munro and Suwanda 1980]), mathematical relationships support the retrieval of elements (e.g., array representation of a heap, see Section 4.3). Explicit data structures must use additional space to store pointers. However, they are more flexible for complex problems. Most programming languages support pointers and basic implicit data structures, such as arrays.

4.1.3.3 Internal vs. External Memory

In a typical computer, there are two levels of memory: internal memory (also called random access memory, i.e., RAM) and external memory (disk). The internal memory is much faster than external memory but has much smaller capacity. Data structures designed to work for data that fit into internal memory may not perform well for large amounts of data that need to be stored in external memory. For large-scale problems, data structures need to be designed that take into account the two levels of memory [Aggarwal and Vitter 1988]. For example, two-level indices such as B-trees [Comer 1979] have been designed to efficiently search in large databases.

4.1.3.4 Space vs. Time

Data structures often exhibit a trade-off between space and time complexity. For example, suppose we want to represent a set of integers in the range $[0, N]$ (e.g., for a set of social security numbers $N = 10^{10} - 1$) such that we can efficiently query whether a given element is in the set, insert an element, or delete an element. Two possible data structures for this problem are an N-element bit array (where the bit in position i indicates the presence of integer i in the set), and a balanced search tree (such as a 2–3 tree or a red–black tree). The bit array has optimal time complexity because it supports queries, insertions, and

deletions in constant time. However, it uses space proportional to the size N of the range, irrespective of the number of elements actually stored. The balanced **search tree** supports queries, insertions, and deletions in logarithmic time but uses optimal space proportional to the current number of elements stored.

4.1.3.5 Theory vs. Practice

A large and ever-growing body of theoretical research on data structures is available, where the performance is measured in asymptotic terms (big-Oh notation). Although asymptotic complexity analysis is an important mathematical subject, it does not completely capture the notion of efficiency of data structures in practical scenarios, where constant factors cannot be disregarded and the difficulty of implementation substantially affects design and maintenance costs. Experimental studies comparing the practical efficiency of data structures for specific classes of problems should be encouraged to bridge the gap between the theory and practice of data structures.

4.1.4 Fundamental Data Structures

The following data structures are ubiquitously used in the description of discrete algorithms, and serve as basic building blocks for realizing more complex data structures. They are covered in detail in the textbooks listed in the "Further Information" section and in the additional references provided.

4.1.4.1 Sequence

A **sequence** is a container that stores elements in a certain linear order, which is imposed by the operations performed. The basic operations supported are retrieving, inserting, and removing an element given its position. Special types of sequences include stacks and queues, where insertions and deletions can be done only at the head or tail of the sequence. The basic realization of sequences are by means of arrays and linked lists. Concatenable queues (see, e.g., Hoffman et al. [1986]) support additional operations such as splitting and splicing, and determining the sequence containing a given element. In external memory, a sequence is typically associated with a file.

4.1.4.2 Priority Queue

A **priority queue** is a container of elements from a totally ordered universe that supports the basic operations of inserting an element and retrieving/removing the largest element. A key application of priority queues is sorting algorithms. A **heap** is an efficient realization of a priority queue that embeds the elements into the ancestor/descendant partial order of a **binary tree**. A heap also admits an implicit realization where the nodes of the tree are mapped into the elements of an array (see Section 4.3). Sophisticated variations of priority queues include min–max heaps, pagodas, deaps, binomial heaps, and Fibonacci heaps. The buffer tree is an efficient external-memory realization of a priority queue.

4.1.4.3 Dictionary

A **dictionary** is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. **Hash tables** provide an efficient implicit realization of a dictionary. Efficient explicit implementations include skip lists [Pugh 1990], tries, and balanced search trees (e.g., **AVL-trees**, red–black trees, 2–3 trees, 2–3–4 trees, weight-balanced trees, biased search trees, splay trees). The technique of fractional cascading [Chazelle and Guibas 1986] speeds up searching for the same element in a collection of dictionaries. In external memory, dictionaries are typically implemented as B-trees and their variations.

The above data structures are widely used in the following application domains:

1. *Graphs and networks:* adjacency matrix, adjacency lists, link-cut tree [Sleator and Tarjan 1983], dynamic expression tree [Cohen and Tamassia 1995], topology tree [Frederickson 1997], SPQR-tree [Di Battista and Tamassia 1996], sparsification tree [Eppstein et al. 1997]. See also, for example, Di Battista et al. [1999], Even [1979], Mehlhorn [1984], and Tarjan [1983].

2. *Text processing:* string, suffix tree, Patricia tree. See, for example, Gonnet and Baeza-Yates [1991].
3. *Geometry and graphics:* binary space partition tree, chain tree, trapezoid tree, range tree, segment tree, interval tree, priority search tree, hull tree, quad tree, R-tree, grid file, metablock tree. For example, see Chiang and Tamassia [1992], Edelsbrunner [1987], Foley et al. [1990], Mehlhorn [1984], Nievergelt and Hinrichs [1993], O'Rourke [1994], and Preparata and Shamos [1985].

4.1.5 Organization of the Chapter

The remainder of this chapter focuses on three fundamental abstract data types: sequences, priority queues, and dictionaries. Examples of efficient data structures and algorithms for implementing them are presented in detail in Section 4.2 through Section 4.4, respectively. Namely, we cover arrays, singly and doubly linked lists, heaps, search trees, (a, b)-trees, AVL-trees, bucket arrays, and hash tables.

4.2 Sequence

4.2.1 Introduction

A *sequence* is a container that stores elements in linear order, which is imposed by the operations performed. The basic operations supported are:

- INSERTRANK: insert an element in a given position.
- REMOVE: remove an element.

Sequences are a basic form of data organization, and are typically used to realize and implement other data types and data structures.

4.2.2 Operations

Using positions (see Section 4.1.1), we can define a more complete repertory of operations for a sequence S:

SIZE(N): return the number of elements N of S.
HEAD(p): assign to p the position of the first element of S; if S is empty, then p is set to null.
TAIL(p): assign to p the position of the last element of S; if S is empty, then p is set to null.
POSITIONRANK(r, p): assign to p the position of the rth element of S; if $r < 1$ or $r > N$, where N is the size of S, then p is set to null.
PREV(p', p''): assign to p'' the position of the element of S preceding the element with position p'; if p' is the position of the first element of S, then p'' is set to null.
NEXT(p', p''): assign to p'' the position of the element of S following the element with position p'; if p' is the position of the last element of S, then p'' is set to null.
INSERTAFTER(e, p', p''): insert element e into S after the element with position p', and return the position p'' of e.
INSERTBEFORE(e, p', p''): insert element e into S before the element with position p', and return the position p'' of e.
INSERTHEAD(e, p): insert element e at the beginning of S, and return the position p of e.
INSERTTAIL(e, p): insert element e at the end of S, and return the position p of e.
INSERTRANK(e, r, p): insert element e in the rth position of S; if $r < 1$ or $r > N + 1$, where N is the current size of S, then p is set to null.
REMOVE(p, e): remove from S and return element e with position p.
MODIFY(p, e): replace with e the element with position p.

Some of the preceding operations can be easily expressed by means of other operations of the repertory. For example, operations HEAD and TAIL can be easily expressed by means of POSITIONRANK and SIZE.

TABLE 4.1 Performance of a Sequence Implemented with an Array

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
POSITIONRANK	$O(1)$
PREV	$O(1)$
NEXT	$O(1)$
INSERTAFTER	$O(N)$
INSERTBEFORE	$O(N)$
INSERTHEAD	$O(N)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(1)$

TABLE 4.2 Performance of a Sequence Implemented with a Singly Linked List

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
POSITIONRANK	$O(N)$
PREV	$O(N)$
NEXT	$O(1)$
INSERTAFTER	$O(1)$
INSERTBEFORE	$O(N)$
INSERTHEAD	$O(1)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(1)$

4.2.3 Implementation with an Array

The simplest way to implement a sequence is to use a (one-dimensional) array, where the ith element of the array stores the ith element of the list, and to keep a variable that stores the size N of the sequence. With this implementation, accessing elements takes $O(1)$ time, whereas insertions and deletions take $O(N)$ time. Table 4.1 shows the time complexity of the implementation of a sequence by means of an array.

4.2.4 Implementation with a Singly Linked List

A sequence can also be implemented with a singly linked list, where each position has a pointer to the next one. We also store the size of the sequence and pointers to the first and last position of the sequence.

With this implementation, accessing elements by rank takes $O(N)$ time because we need to traverse the list, whereas some insertions and deletions take $O(1)$ time. Table 4.2 shows the time complexity of the implementation of a sequence by means of a singly linked list.

4.2.5 Implementation with a Doubly Linked List

Better performance can be achieved, at the expense of using additional space, by implementing a sequence with a doubly linked list, where each position has pointers to the next and previous positions. We also

TABLE 4.3 Performance of a Sequence Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
POSITIONRANK	$O(N)$
PREV	$O(1)$
NEXT	$O(1)$
INSERTAFTER	$O(1)$
INSERTBEFORE	$O(1)$
INSERTHEAD	$O(1)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(1)$
MODIFY	$O(1)$

store the size of the sequence and pointers to the first and last positions of the sequence. Table 4.3 shows the time complexity of the implementation of sequence by means of a doubly linked list.

4.3 Priority Queue

4.3.1 Introduction

A priority queue is a container of elements from a totally ordered universe that supports the following two basic operations:

1. INSERT: insert an element into the priority queue.
2. REMOVEMAX: remove the largest element from the priority queue.

Here are some simple applications of a priority queue:

- *Scheduling.* A scheduling system can store the tasks to be performed into a priority queue, and select the task with highest priority to be executed next.
- *Sorting.* To sort a set of N elements, we can insert them one at a time into a priority queue by means of N INSERT operations, and then retrieve them in decreasing order by means of N REMOVEMAX operations. This two-phase method is the paradigm of several popular sorting algorithms, including *selection sort*, *insertion sort*, and *heap-sort*.

4.3.2 Operations

Using locators, we can define a more complete repertory of operations for a priority queue Q:

SIZE(N): return the current number of elements N in Q.
MAX(c): return a locator c to the maximum element of Q.
INSERT(e, c): insert element e into Q and return a locator c to e.
REMOVE(c, e): remove from Q and return element e with locator c.
REMOVEMAX(e): remove from Q and return the maximum element e from Q.
MODIFY(c, e): replace with e the element with locator c.

Note that operation REMOVEMAX(e) is equivalent to MAX(c) followed by REMOVE(c, e).

TABLE 4.4 Performance of a Priority Queue Realized by an Unsorted Sequence, Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
MAX	$O(N)$
INSERT	$O(1)$
REMOVE	$O(1)$
REMOVEMAX	$O(N)$
MODIFY	$O(1)$

TABLE 4.5 Performance of a Priority Queue Realized by a Sorted Sequence, Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(1)$
REMOVEMAX	$O(1)$
MODIFY	$O(N)$

4.3.3 Realization with a Sequence

We can realize a priority queue by reusing and extending the sequence abstract data type (see Section 4.2). Operations SIZE, MODIFY, and REMOVE correspond to the homonymous sequence operations.

4.3.3.1 Unsorted Sequence

We can realize INSERT by an INSERTHEAD or an INSERTTAIL, which means that the sequence is not kept sorted. Operation MAX can be performed by scanning the sequence with an iteration of NEXT operations, keeping track of the maximum element encountered. Finally, as observed earlier, operation REMOVEMAX is a combination of MAX and REMOVE. Table 4.4 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly linked list. In the table we denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

4.3.3.2 Sorted Sequence

An alternative implementation uses a sequence that is kept sorted. In this case, operation MAX corresponds to simply accessing the last element of the sequence. However, operation INSERT now requires scanning the sequence to find the appropriate position to insert the new element. Table 4.5 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly linked list. In the table we denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

Realizing a priority queue with a sequence, sorted or unsorted, has the drawback that some operations require linear time in the worst case. Hence, this realization is not suitable in many applications where fast running times are sought for all the priority queue operations.

4.3.3.3 Sorting

For example, consider the sorting application (see the first introduction to this section). We have a collection of N elements from a totally ordered universe, and we want to sort them using a priority queue Q. We

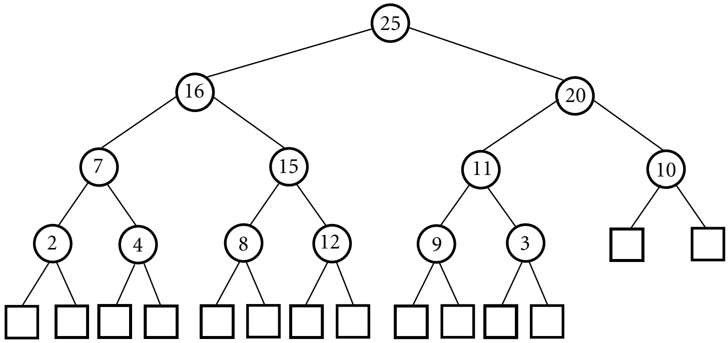


FIGURE 4.1 Example of a heap storing 13 elements.

assume that each element uses $O(1)$ space, and any two elements can be compared in $O(1)$ time. If we realize Q with an unsorted sequence, then the first phase (inserting the N elements into Q) takes $O(N)$ time. However, the second phase (removing N times the maximum element) takes time

$$O\left(\sum_{i=1}^{N} i\right) = O(N^2)$$

Hence, the overall time complexity is $O(N^2)$. This sorting method is known as *selection sort.*

However, if we realize the priority queue with a sorted sequence, then the first phase takes time

$$O\left(\sum_{i=1}^{N} i\right) = O(N^2)$$

while the second phase takes time $O(N)$. Again, the overall time complexity is $O(N^2)$. This sorting method is known as *insertion sort.*

4.3.4 Realization with a Heap

A more sophisticated realization of a priority queue uses a data structure called a *heap*. A heap is a binary tree T whose internal nodes each store one element from a totally ordered universe, with the following properties (see Figure 4.1):

Level property. All of the levels of T are full, except possibly for the bottommost level, which is left filled.

Partial order property. Let μ be a node of T distinct from the root, and let ν be the parent of μ; then the element stored at μ is less than or equal to the element stored at ν.

The leaves of a heap do not store data and serve only as placeholders. The level property implies that heap T is a minimum-height binary tree. More precisely, if T stores N elements and has height h, then each level i with $0 \leq i \leq h-2$ stores exactly 2^i elements, whereas level $h-1$ stores between 1 and 2^{h-1} elements. Note that level h contains only leaves. We have

$$2^{h-1} = 1 + \sum_{i=0}^{h-2} 2^i \leq N \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

from which we obtain:

$$\log_2(N+1) \leq h \leq 1 + \log_2 N$$

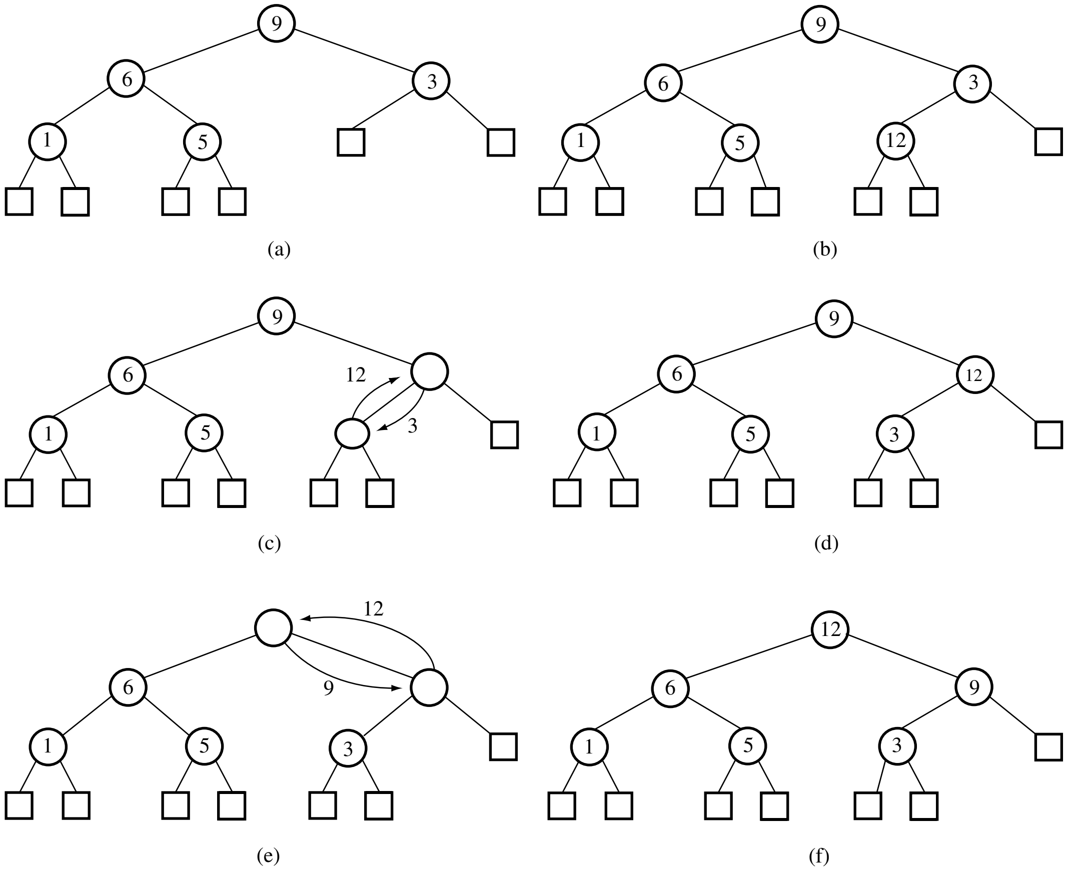


FIGURE 4.2 Operation INSERT in a heap.

Now we show how to perform the various priority queue operations by means of a heap T. We denote with $x(\mu)$ the element stored at an internal node μ of T. We denote with ρ the root of T. We call the *last node* of T the rightmost internal node of the bottommost internal level of T.

By storing a counter that keeps track of the current number of elements, SIZE consists of simply returning the value of the counter. By the partial order property, the maximum element is stored at the root and, hence, operation MAX can be performed by accessing node ρ.

4.3.4.1 Operation INSERT

To insert an element e into T, we add a new internal node μ to T such that μ becomes the new last node of T, and set $x(\mu) = e$. This action ensures that the level property is satisfied, but may violate the partial order property. Hence, if $\mu \neq \rho$, we compare $x(\mu)$ with $x(\nu)$, where ν is the parent of μ. If $x(\mu) > x(\nu)$, then we need to restore the partial order property, which can be locally achieved by exchanging the elements stored at μ and ν. This causes the new element e to move up one level. Again, the partial order property may be violated, and we may have to continue moving up the new element e until no violation occurs. In the worst case, the new element e moves up to the root ρ of T by means of $O(\log N)$ exchanges. The upward movement of element e by means of exchanges is conventionally called *upheap*.

An example of a sequence of insertions into a heap is shown in Figure 4.2.

4.3.4.2 Operation REMOVEMAX

To remove the maximum element, we cannot simply delete the root of T, because this would disrupt the binary tree structure. Instead, we access the last node λ of T, copy its element e to the root by setting $x(\rho) = x(\lambda)$, and delete λ. We have preserved the level property, but we may have violated the partial order property. Hence, if ρ has at least one nonleaf child, we compare $x(\rho)$ with the maximum element $x(\sigma)$ stored at a child of ρ. If $x(\rho) < x(\sigma)$, then we need to restore the partial order property, which can be locally achieved by exchanging the elements stored at ρ and σ. Again, the partial order property may be violated, and we continue moving down element e until no violation occurs. In the worst case, element e moves down to the bottom internal level of T by means of $O(\log N)$ exchanges. The downward movement of element e by means of exchanges is conventionally called *downheap*.

An example of operation REMOVEMAX in a heap is shown in Figure 4.3.

4.3.4.3 Operation REMOVE

To remove an arbitrary element of heap T, we cannot simply delete its node μ, because this would disrupt the binary tree structure. Instead, we proceed as before and delete the last node of T after copying to μ its element e. We have preserved the level property, but we may have violated the partial order property, which can be restored by performing either upheap or downheap.

Finally, after modifying an element of heap T, if the partial order property is violated, we just need to perform either upheap or downheap.

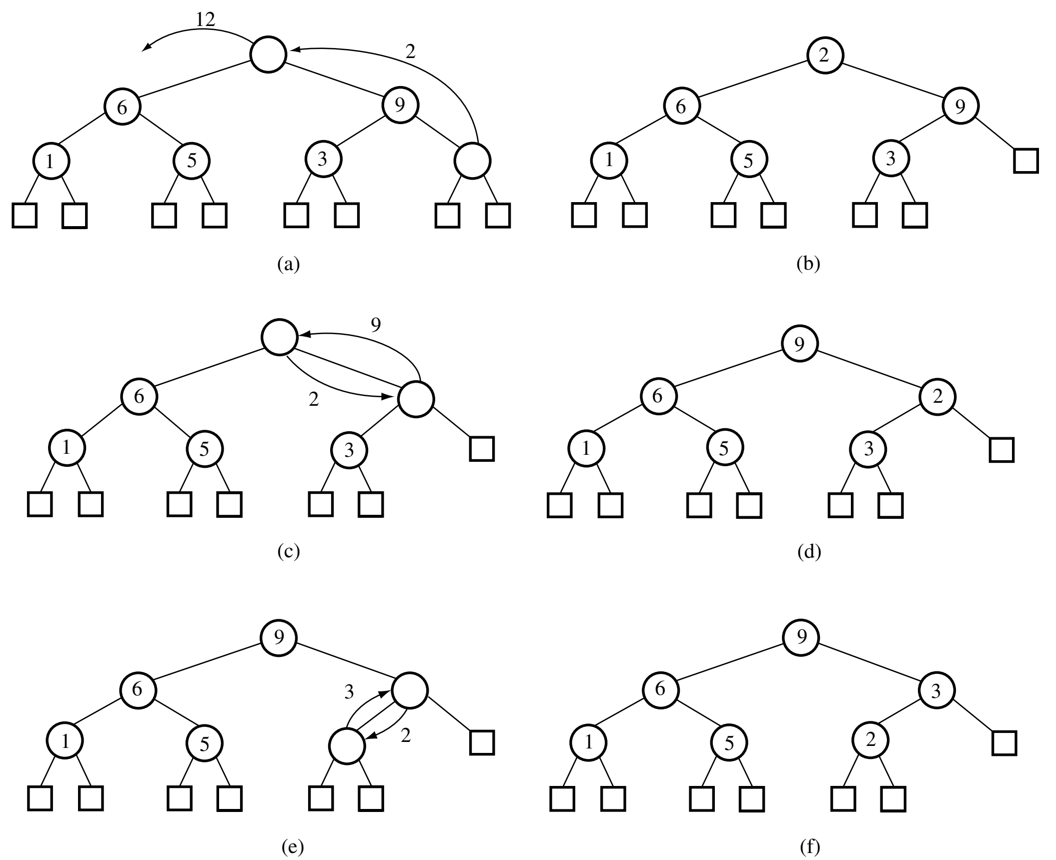


FIGURE 4.3 Operation REMOVEMAX in a heap.

TABLE 4.6 Performance of a Priority Queue Realized by a Heap, Implemented with a Suitable Binary Tree Data Structure

Operation	Time
SIZE	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
REMOVEMAX	$O(\log N)$
MODIFY	$O(\log N)$

4.3.4.4 Time Complexity

Table 4.6 shows the time complexity of the realization of a priority queue by means of a heap. In the table we denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$. We assume that the heap is itself realized by a data structure for binary trees that supports $O(1)$-time access to the children and parent of a node. For instance, we can implement the heap explicitly with a linked structure (with pointers from a node to its parents and children), or implicitly with an array (where node i has children $2i$ and $2i + 1$). Let N be the number of elements in a priority queue Q realized with a heap T at the time an operation is performed. The time bounds of Table 4.6 are based on the following facts:

- In the worst case, the time complexity of upheap and downheap is proportional to the height of T.
- If we keep a pointer to the last node of T, we can update this pointer in time proportional to the height of T in operations INSERT, REMOVE, and REMOVEMAX, as illustrated in Figure 4.4.
- The height of heap T is $O(\log N)$.

The $O(N)$ space complexity bound for the heap is based on the following facts:

- The heap has $2N + 1$ nodes (N internal nodes and $N + 1$ leaves).
- Every node uses $O(1)$ space.
- In the array implementation, because of the level property, the array elements used to store heap nodes are in the contiguous locations 1 through $2N - 1$.

Note that we can reduce the space requirement by a constant factor implementing the leaves of the heap with null objects, such that only the internal nodes have space associated with them.

4.3.4.5 Sorting

Realizing a priority queue with a heap has the advantage that all of the operations take $O(\log N)$ time, where N is the number of elements in the priority queue at the time the operation is performed. For example, in the sorting application (see Section 4.3.1), both the first phase (inserting the N elements) and the second phase (removing N times the maximum element) take time

$$O\left(\sum_{i=1}^{N} \log i\right) = O(N \log N)$$

Hence, sorting with a priority queue realized with a heap takes $O(N \log N)$ time. This sorting method is known as *heap sort*, and its performance is considerably better than that of selection sort and insertion sort (see Section 4.3.3.3), where the priority queue is realized as a sequence.

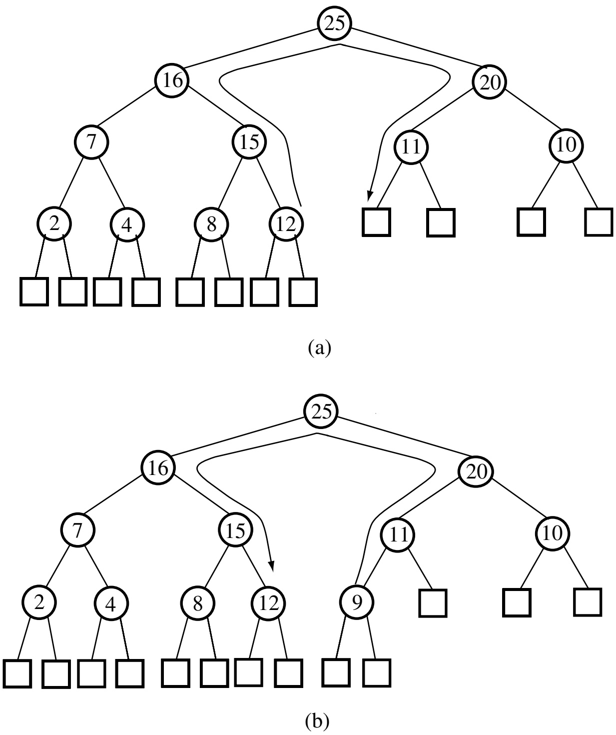


FIGURE 4.4 Update of the pointer to the last node: (a) INSERT and (b) REMOVE or REMOVEMAX.

4.3.5 Realization with a Dictionary

A priority queue can be easily realized with a dictionary (see Section 4.4). Indeed, all of the operations in the priority queue repertory are supported by a dictionary. To achieve $O(1)$ time for operation MAX, we can store the locator of the maximum element in a variable, and recompute it after an update operation. This realization of a priority queue with a dictionary has the same asymptotic complexity bounds as the realization with a heap, provided the dictionary is suitably implemented, for example, with an (a, b)-tree (see section "Realization with an (a, b)-tree") or an AVL-tree (see section "Realization with an AVL-tree"). However, a heap is simpler to program than an (a, b)-tree or an AVL-tree.

4.4 Dictionary

A dictionary is a container of elements from a totally ordered universe that supports the following basic operations:

- FIND: search for an element.
- INSERT: insert an element.
- REMOVE: delete an element.

A major application of dictionaries is database systems.

4.4.1 Operations

In the most general setting, the elements stored in a dictionary are pairs (x, y), where x is the *key* giving the ordering of the elements and y is the auxiliary information. For example, in a database storing student

records, the key could be the student's last name, and the auxiliary information the student's transcript. It is convenient to augment the ordered universe of keys with two *special keys* ($+\infty$ and $-\infty$) and assume that each dictionary has, in addition to its *regular elements*, two *special elements*, with keys $+\infty$ and $-\infty$, respectively. For simplicity, we will also assume that no two elements of a dictionary have the same key. An insertion of an element with the same key as that of an existing element will be rejected by returning a null locator.

Using locators (see Section 4.1), we can define a more complete repertory of operations for a dictionary D:

SIZE(N): return the number of regular elements N of D.

FIND(x, c): if D contains an element with key x, assign to c a locator to such as an element; otherwise; set c equal to a null locator.

LOCATEPREV(x, c): assign to c a locator to the element of D with the largest key less than or equal to x; if x is smaller than all of the keys of the regular elements, then c is a locator to the special element with key $-\infty$; if $x = -\infty$, then c is a null locator.

LOCATENEXT(x, c): assign to c a locator to the element of D with the smallest key greater than or equal to x; if x is larger than all of the keys of the regular elements, then c is a locator to the special element with key $+\infty$; then, if $x = +\infty$, c is a null locator.

PREV(c', c''): assign to c'' a locator to the element of D with the largest key less than that of the element with locator c'; if the key of the element with locator c' is smaller than all of the keys of the regular elements, then this operation returns a locator to the special element with key $-\infty$.

NEXT(c', c''): assign to c'' a locator to the element of D with the smallest key larger than that of the element with locator c'; if the key of the element with locator c' is larger than all of the keys of the regular elements, then this operation returns a locator to the special element with key $+\infty$.

MIN(c): assign to c a locator to the regular element of D with minimum key; if D has no regular elements, then c is a null locator.

MAX(c): assign to c a locator to the regular element of D with maximum key; if D has no regular elements, then c is a null locator.

INSERT(e, c): insert element e into D, and return a locator c to e; if there is already an element with the same key as e, then this operation returns a null locator.

REMOVE(c, e): remove from D and return element e with locator c.

MODIFY(c, e): replace with e the element with locator c.

Some of these operations can be easily expressed by means of other operations of the repertory. For example, operation FIND is a simple variation of LOCATEPREV or LOCATENEXT.

4.4.2 Realization with a Sequence

We can realize a dictionary by reusing and extending the sequence abstract data type (see Section 4.2). Operations SIZE, INSERT, and REMOVE correspond to the homonymous sequence operations.

4.4.2.1 Unsorted Sequence

We can realize INSERT by an INSERTHEAD or an INSERTTAIL, which means that the sequence is not kept sorted. Operation FIND(x, c) can be performed by scanning the sequence with an iteration of NEXT operations, until we either find an element with key x, or we reach the end of the sequence. Table 4.7 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly linked list. In the table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.2.2 Sorted Sequence

We can also use a sorted sequence to realize a dictionary. Operation INSERT now requires scanning the sequence to find the appropriate position to insert the new element. However, in a FIND operation, we can stop scanning the sequence as soon as we find an element with a key larger than the search key.

TABLE 4.7 Performance of a Dictionary Realized by an Unsorted Sequence, Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
FIND	$O(N)$
LOCATEPREV	$O(N)$
LOCATENEXT	$O(N)$
NEXT	$O(N)$
PREV	$O(N)$
MIN	$O(N)$
MAX	$O(N)$
INSERT	$O(1)$
REMOVE	$O(1)$
MODIFY	$O(1)$

TABLE 4.8 Performance of a Dictionary Realized by a Sorted Sequence, Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
FIND	$O(N)$
LOCATEPREV	$O(N)$
LOCATENEXT	$O(N)$
NEXT	$O(1)$
PREV	$O(1)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(1)$
MODIFY	$O(N)$

Table 4.8 shows the time complexity of this realization by a sorted sequence, assuming that the sequence is implemented with a doubly linked list. In the table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.2.3 Sorted Array

We can obtain a different performance trade-off by implementing the sorted sequence by means of an array, which allows constant-time access to any element of the sequence given its position. Indeed, with this realization we can speed up operation FIND(x, c) using the *binary search* strategy, as follows. If the dictionary is empty, we are done. Otherwise, let N be the current number of elements in the dictionary. We compare the search key k with the key x_m of the middle element of the sequence, that is, the element at position $\lfloor N/2 \rfloor$. If $x = x_m$, we have found the element. Else, we recursively search in the subsequence of the elements preceding the middle element if $x < x_m$, or following the middle element if $x > x_m$. At each recursive call, the number of elements of the subsequence being searched halves. Hence, the number of sequence elements accessed and the number of comparisons performed by binary search is $O(\log N)$. While searching takes $O(\log N)$ time, inserting or deleting elements now takes $O(N)$ time.

TABLE 4.9 Performance of a Dictionary Realized by a Sorted Sequence, Implemented with an Array

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
NEXT	$O(1)$
PREV	$O(1)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(N)$

Table 4.9 shows the performance of a dictionary realized with a sorted sequence, implemented with an array. In the table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.3 Realization with a Search Tree

A *search tree* for elements of the type (x, y), where x is a key from a totally ordered universe, is a rooted ordered tree T such that:

- Each internal node of T has at least two children and stores a nonempty set of elements.
- A node μ of T with d children $\mu_1, \ldots, \mu_d$ stores $d - 1$ elements $(x_1, y_1) \cdots (x_{d-1}, y_{d-1})$, where $x_1 \leq \cdots \leq x_{d-1}$.
- For each element (x, y) stored at a node in the subtree of T rooted at μ_i, we have $x_{i-1} \leq x \leq x_i$, where $x_0 = -\infty$ and $x_d = +\infty$.

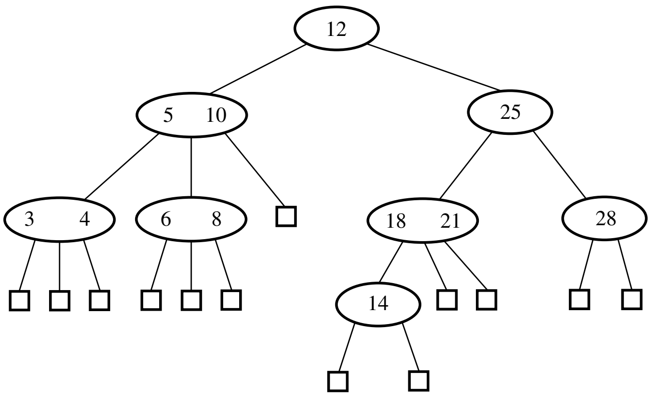
In a search tree, each internal node stores a nonempty collection of keys, whereas the leaves do not store any key and serve only as placeholders. An example search tree is shown in Figure 4.5a. A special type of search tree is a *binary search tree*, where each internal node stores one key and has two children.

We will recursively describe the realization of a dictionary D by means of a search tree T because we will use dictionaries to implement the nodes of T. Namely, an internal node μ of T with children $\mu_1, \ldots, \mu_d$ and elements $(x_1, y_1) \cdots (x_{d-1}, y_{d-1})$ is equipped with a dictionary $D(\mu)$ whose regular elements are the pairs $(x_i, (y_i, \mu_i))$, $i = 1, \ldots, d - 1$ and whose special element with key $+\infty$ is $(+\infty, (\cdot, \mu_d))$. A regular element (x, y) stored in D is associated with a regular element $(x, (y, \nu))$ stored in a dictionary $D(\mu)$, for some node μ of T. See the example in Figure 4.5b.

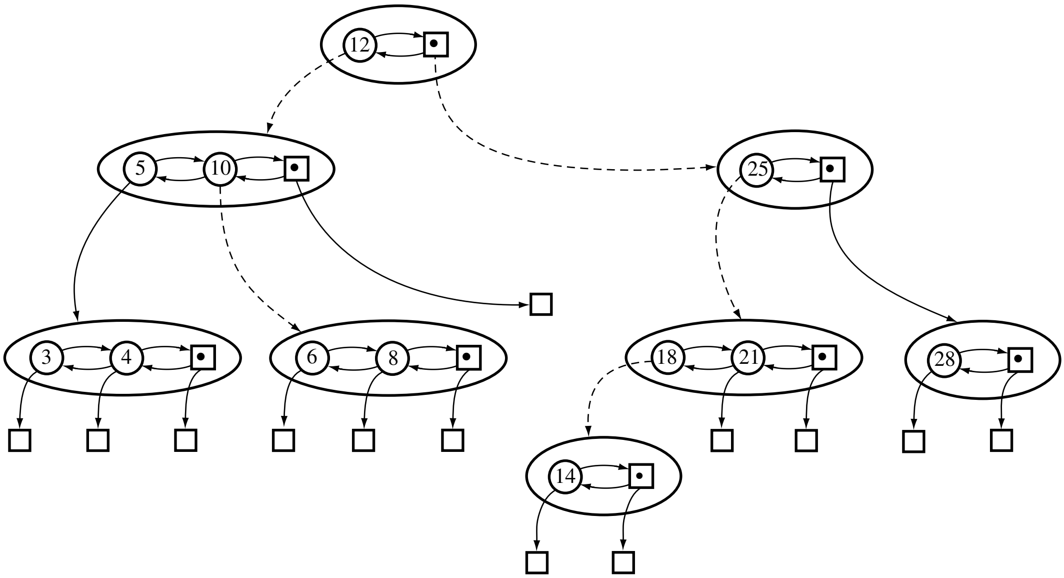
4.4.3.1 Operation FIND

Operation FIND(x, c) on dictionary D is performed by means of the following recursive method for a node μ of T, where μ is initially the root of T [see Figure 4.5b]. We execute LOCATENEXT(x, c') on dictionary $D(\mu)$ and let $(x', (y', \nu))$ be the element pointed by the returned locator c'. We have three cases:

1. Case $x = x'$: we have found x and return locator c to (x', y').
2. Case $x \neq x'$ and ν is a leaf: we have determined that x is not in D and return a null locator c.
3. Case $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.



(a)



(b)

FIGURE 4.5 Realization of a dictionary by means of a search tree: (a) a search tree T, (b) realization of the dictionaries at the nodes of T by means of sorted sequences. The search paths for elements 9 (unsuccessful search) and 14 (successful search) are shown with dashed lines.

4.4.3.2 Operation Insert

Operations LocatePrev, LocateNext, and Insert can be performed with small variations of the previously described method. For example, to perform operation Insert(e, c), where $e = (x, y)$, we modify the previous cases as follows (see Figure 4.6):

1. Case $x = x'$: an element with key x already exists, and we return a null locator.
2. Case $x \neq x'$ and ν is a leaf: we create a new leaf node λ, insert a new element $(x, (y, \lambda))$ into $D(\mu)$, and return a locator c to (x, y).
3. Case $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.

Note that new elements are inserted at the bottom of the search tree.

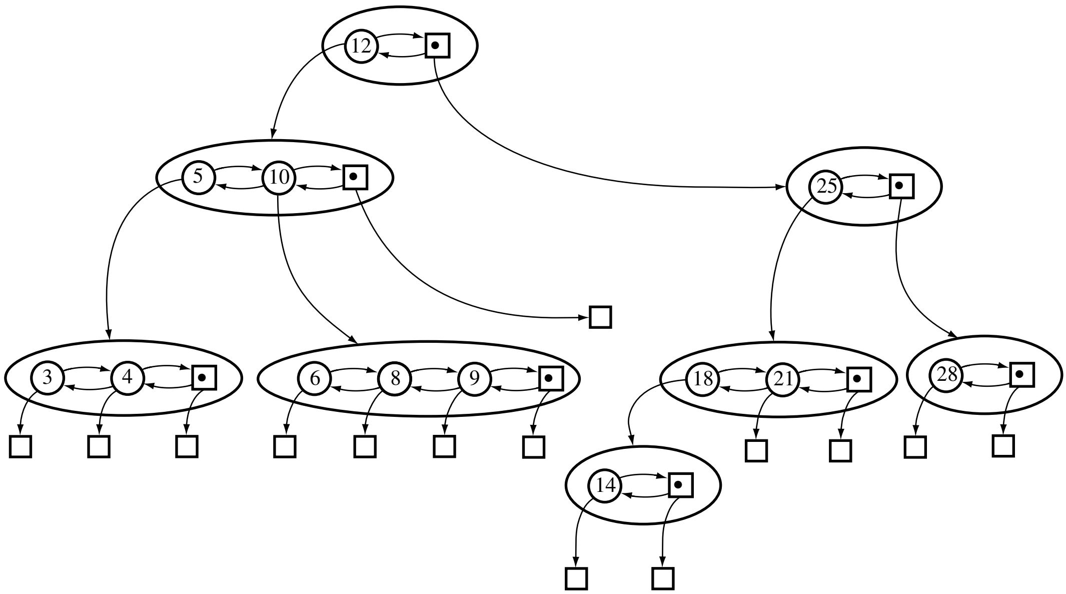


FIGURE 4.6 Insertion of element 9 into the search tree of Figure 4.5.

4.4.3.3 Operation Remove

Operation REMOVE(e, c) is more complex (see Figure 4.7). Let the associated element of $e = (x, y)$ in T be $(x, (y, \nu))$, stored in dictionary $D(\mu)$ of node μ:

- If node ν is a leaf, we simply delete element $(x, (y, \nu))$ from $D(\mu)$.
- Else (ν is an internal node), we find the successor element $(x', (y', \nu'))$ of $(x, (y, \nu))$ in $D(\mu)$ with a NEXT operation in $D(\mu)$. (1) If ν' is a leaf, we replace ν' with ν, that is, change element $(x', (y', \nu'))$ to $(x', (y', \nu))$, and delete element $(x, (y, \nu))$ from $D(\mu)$. (2) Else (ν' is an internal node), while the leftmost child ν'' of ν' is not a leaf, we set $\nu' = \nu''$. Let $(x'', (y'', \nu''))$ be the first element of $D(\nu')$ (node ν'' is a leaf). We replace $(x, (y, \nu))$ with $(x'', (y'', \nu))$ in $D(\mu)$ and delete $(x'', (y'', \nu''))$ from $D(\nu')$.

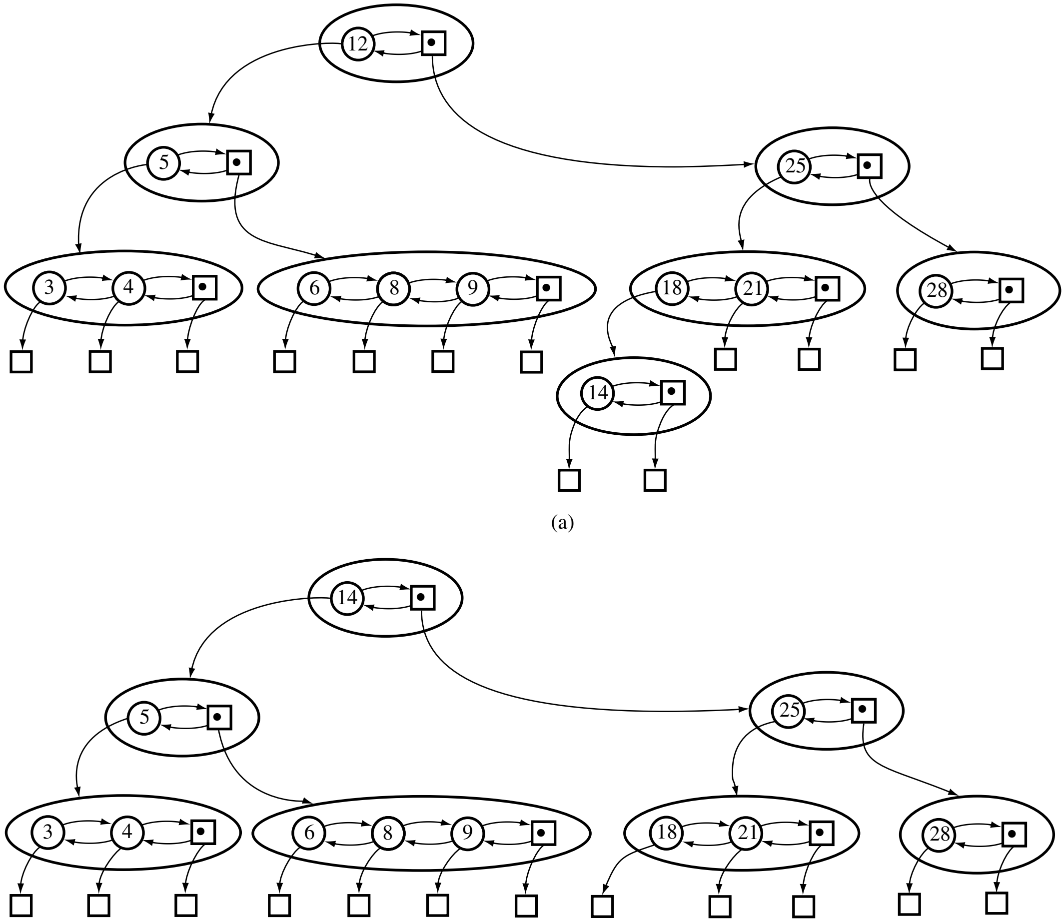
The listed actions may cause dictionary $D(\mu)$ or $D(\nu')$ to become empty. If this happens, say for $D(\mu)$ and μ is not the root of T, we need to remove node μ. Let $(+\infty, (\cdot, \kappa))$ be the special element of $D(\mu)$ with key $+\infty$, and let $(z, (w, \mu))$ be the element pointing to μ in the parent node π of μ. We delete node μ and replace $(z, (w, \mu))$ with $(z, (w, \kappa))$ in $D(\pi)$.

Note that if we start with an initially empty dictionary, a sequence of insertions and deletions performed with the described methods yields a search tree with a single node. In the next sections, we show how to avoid this behavior by imposing additional conditions on the structure of a search tree.

4.4.4 Realization with an (a, b)-Tree

An (a, b)-*tree*, where a and b are integer constants such that $2 \leq a \leq (b + 1)/2$, is a a search tree T with the following additional restrictions:

Level property. All of the levels of T are full, that is, all of the leaves are at the same depth.
Size property. Let μ be an internal node of T, and d be the number of children of μ; if μ is the root of T, then $d \geq 2$, else $a \leq d \leq b$.



(a)

(b)

FIGURE 4.7 (a) Deletion of element 10 from the search tree of Figure 4.6. (b) Deletion of element 12 from the search tree of part a.

The height of an (a, b)-tree storing N elements is $O(\log_a N) = O(\log N)$. Indeed, in the worst case, the root has two children and all of the other internal nodes have a children.

The realization of a dictionary with an (a, b)-tree extends that with a search tree. Namely, the implementation of operations INSERT and REMOVE need to be modified in order to preserve the level and size properties. Also, we maintain the current size of the dictionary, and pointers to the minimum and maximum regular elements of the dictionary.

4.4.4.1 Insertion

The implementation of operation INSERT for search trees given earlier in this section adds a new element to the dictionary $D(\mu)$ of an existing node μ of T. Because the structure of the tree is not changed, the level property is satisfied. However, if $D(\mu)$ had the maximum allowed size $b - 1$ before insertion (recall that the size of $D(\mu)$ is one less than the number of children of μ), then the size property is violated at μ because $D(\mu)$ has now size b. To remedy this *overflow* situation, we perform the following *node split* (see Figure 4.8):

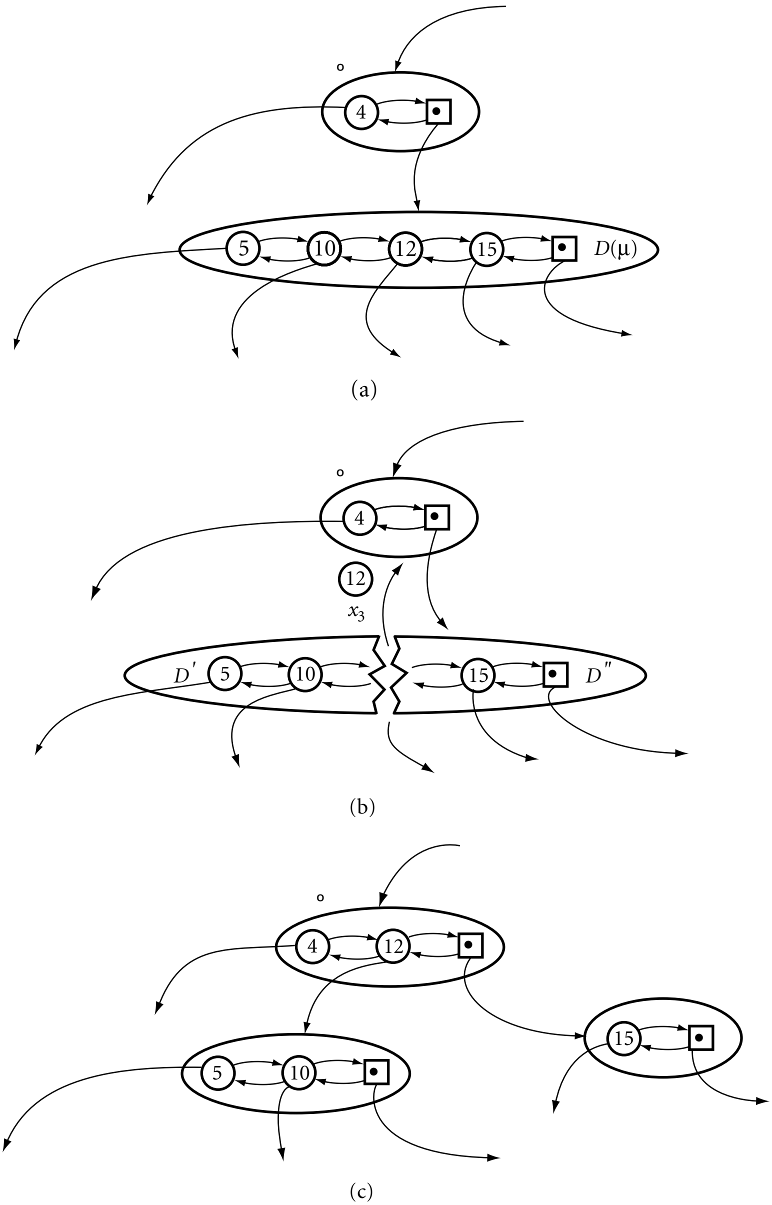


FIGURE 4.8 Example of node split in a 2–4 tree: (a) initial configuration with an overflow at node μ, (b) split of the node μ into μ' and μ'' and insertion of the median element into the parent node π, and (c) final configuration.

- Let the special element of $D(\mu)$ be $(+\infty, (\cdot, \mu_{b+1}))$. Find the median element of $D(\mu)$, that is, the element $e_i = (x_i, (y_i, \mu_i))$ such that $i = \lceil (b+1)/2 \rceil$).
- Split $D(\mu)$ into: (1) dictionary D' containing the $\lceil (b-1)/2 \rceil$ regular elements $e_j = (x_j, (y_j, \mu_j))$, $j = 1 \cdots i-1$ and the special element $(+\infty, (\cdot, \mu_i))$; (2) element e; and (3) dictionary D'', containing the $\lfloor (b-1)/2 \rfloor$ regular elements $e_j = (x_j, (y_j, \mu_j))$, $j = i+1 \cdots b$ and the special element $(+\infty, (\cdot, \mu_{b+1}))$.
- Create a new tree node κ, and set $D(\kappa) = D'$. Hence, node κ has children $\mu_1 \cdots \mu_i$.

- Set $D(\mu) = D''$. Hence, node μ has children $\mu_{i+1} \cdots \mu_{b+1}$.
- If μ is the root of T, create a new node π with an empty dictionary $D(\pi)$. Else, let π be the parent of μ.
- Insert element $(x_i, (y_i, \kappa))$ into dictionary $D(\pi)$.

After a node split, the level property is still verified. Also, the size property is verified for all of the nodes of T, except possibly for node π. If π has $b + 1$ children, we repeat the node split for $\mu = \pi$. Each time we perform a node split, the possible violation of the size property appears at a higher level in the tree. This guarantees the termination of the algorithm for the INSERT operation. We omit the description of the simple method for updating the pointers to the minimum and maximum regular elements.

4.4.4.2 Deletion

The implementation of operation REMOVE for search trees given earlier in this section removes an element from the dictionary $D(\mu)$ of an existing node μ of T. Because the structure of the tree is not changed, the level property is satisfied. However, if μ is not the root, and $D(\mu)$ had the minimum allowed size $a - 1$ before deletion (recall that the size of the dictionary is one less than the number of children of the node), then the size property is violated at μ because $D(\mu)$ has now size $a - 2$. To remedy this *underflow* situation, we perform the following *node merge* (see Figure 4.9 and Figure 4.10):

- If μ has a right sibling, then let μ'' be the right sibling of μ and $\mu' = \mu$; else, let μ' be the left sibling of μ and $\mu'' = \mu$. Let $(+\infty, (\cdot, \nu))$ be the special element of $D(\mu')$.
- Let π be the parent of μ' and μ''. Remove from $D(\pi)$ the regular element $(x, (y, \mu'))$ associated with μ'.
- Create a new dictionary D containing the regular elements of $D(\mu')$ and $D(\mu'')$, regular element $(x, (y, \nu))$, and the special element of $D(\mu'')$.
- Set $D(\mu'') = D$, and destroy node μ'.
- If μ'' has more than b children, perform a node split at μ''.

After a node merge, the level property is still verified. Also, the size property is verified for all the nodes of T, except possibly for node π. If π is the root and has one child (and thus an empty dictionary), we remove node π. If π is not the root and has fewer than $a - 1$ children, we repeat the node merge for $\mu = \pi$. Each time we perform a node merge, the possible violation of the size property appears at a higher level in the tree. This guarantees the termination of the algorithm for the REMOVE operation. We omit the description of the simple method for updating the pointers to the minimum and maximum regular elements.

4.4.4.3 Complexity

Let T be an (a, b)-tree storing N elements. The height of T is $O(\log_a N) = O(\log N)$. Each dictionary operation affects only the nodes along a root-to-leaf path. We assume that the dictionaries at the nodes of T are realized with sequences. Hence, processing a node takes $O(b) = O(1)$ time. We conclude that each operation takes $O(\log N)$ time.

Table 4.10 shows the performance of a dictionary realized with an (a, b)-tree. In the table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.5 Realization with an AVL-Tree

An *AVL-tree* is a search tree T with the following additional restrictions:

Binary property. T is a binary tree, that is, every internal node has two children (left and right child), and stores one key.

Balance property. For every internal node μ, the heights of the subtrees rooted at the children of μ differ at most by one.

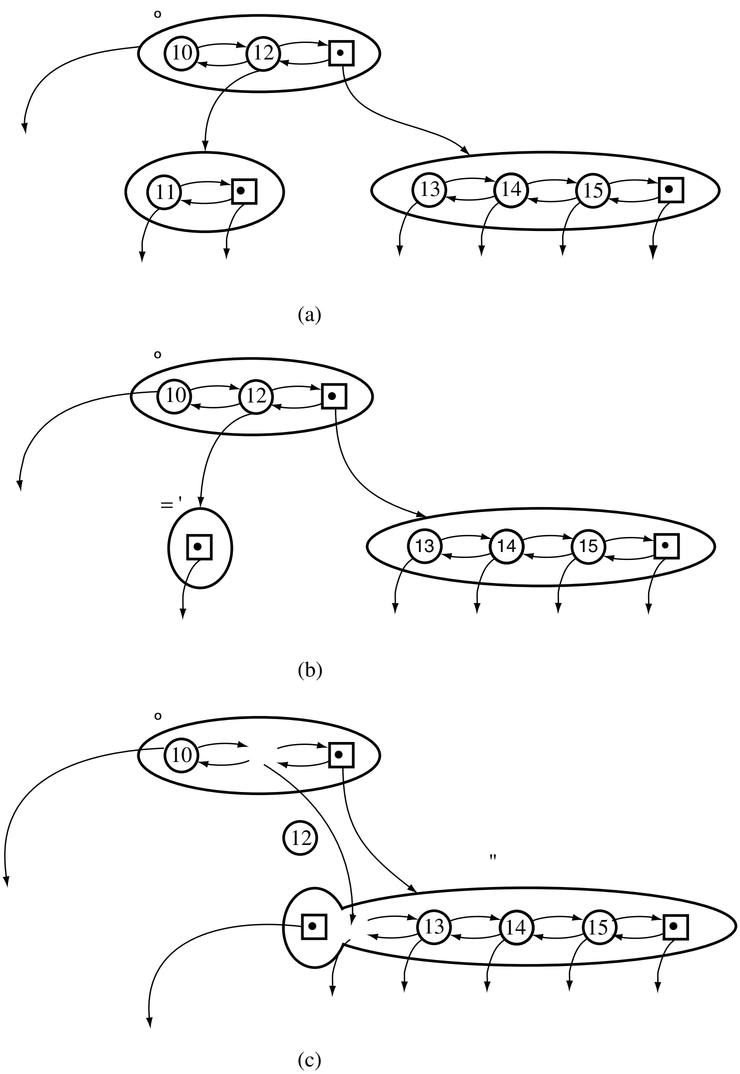


FIGURE 4.9 Example of node merge in a 2–4 tree: (a) initial configuration, (b) the removal of an element from dictionary $D(\mu)$ causes an underflow at node μ, and (c) merging node $\mu = \mu'$ into its sibling μ''.

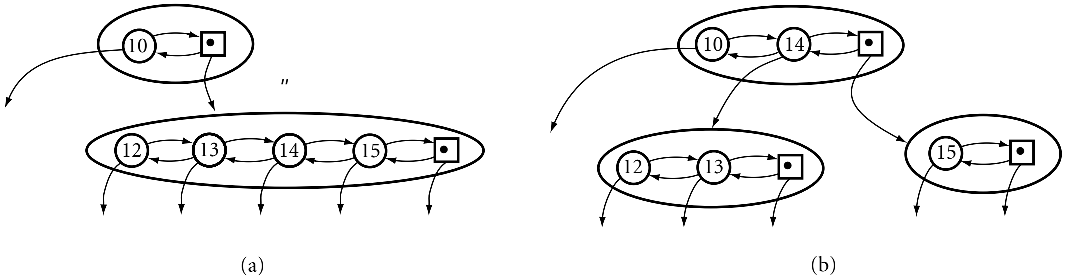


FIGURE 4.10 Example of subsequent node merge in a 2–4 tree: (a) overflow at node μ'' and (b) final configuration after splitting node μ''.

TABLE 4.10 Performance of a Dictionary Realized by an (a, b)-Tree

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
NEXT	$O(\log N)$
PREV	$O(\log N)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
MODIFY	$O(\log N)$

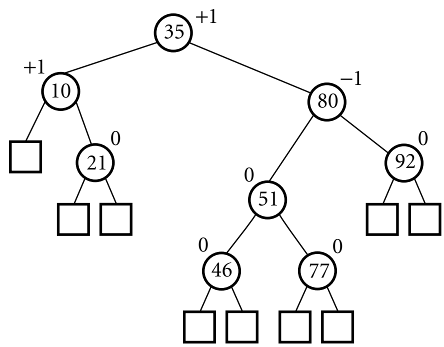


FIGURE 4.11 Example of AVL-tree storing nine elements. The keys are shown inside the nodes, and the balance factors (see subsequent section on rebalancing) are shown next to the nodes.

An example of AVL-tree is shown in Figure 4.11. The height of an AVL-tree storing N elements is $O(\log N)$. This can be shown as follows. Let N_h be the minimum number of elements stored in an AVL-tree of height h. We have $N_0 = 0$, $N_1 = 1$, and

$$N_h = 1 + N_{h-1} + N_{h-2}, \quad \text{for } h \geq 2$$

The preceding recurrence relation defines the well-known Fibonacci numbers. Hence, $N_h = \Omega(\phi^N)$, where $\phi = (1 + \sqrt{5})/2 = 1.6180\cdots$ is the golden ratio.

The realization of a dictionary with an AVL-tree extends that with a search tree. Namely, the implementation of operations INSERT and REMOVE must be modified to preserve the binary and balance properties after an insertion or deletion.

4.4.5.1 Insertion

The implementation of INSERT for search trees given earlier in this section adds the new element to an existing node. This violates the binary property, and hence cannot be done in an AVL-tree. Hence, we modify the three cases of the INSERT algorithm for search trees as follows:

- Case $x = x'$: an element with key x already exists, and we return a null locator c.
- Case $x \neq x'$ and ν is a leaf: we replace ν with a new internal node κ with two leaf children, store element (x, y) in κ, and return a locator c to (x, y).
- Case $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.

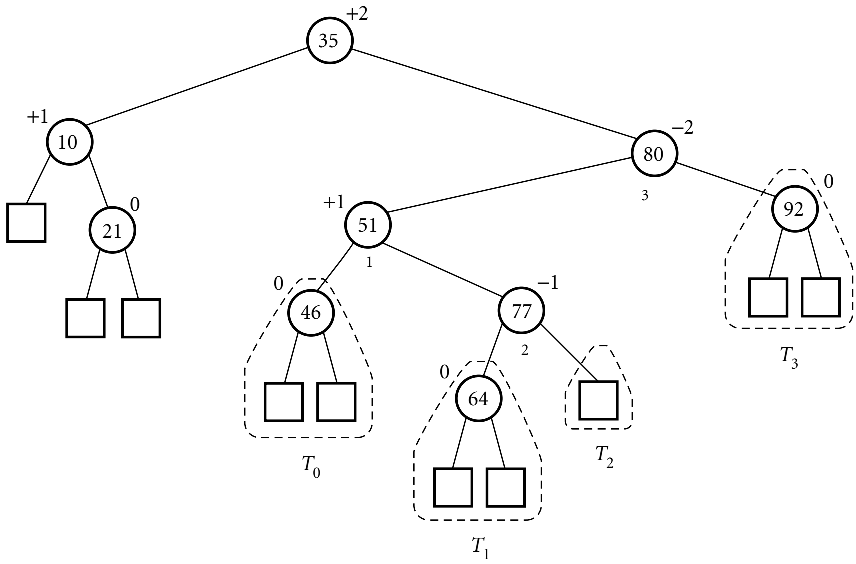


FIGURE 4.12 Insertion of an element with key 64 into the AVL-tree of Figure 4.11. Note that two nodes (with balance factors +2 and −2) have become unbalanced. The dashed lines identify the subtrees that participate in the rebalancing, as illustrated in Figure 4.14.

We have preserved the binary property. However, we may have violated the balance property because the heights of some subtrees of T have increased by one. We say that a node is balanced if the difference between the heights of its subtrees is −1, 0, or 1, and is unbalanced otherwise. The unbalanced nodes form a (possibly empty) subpath of the path from the new internal node κ to the root of T. See the example of Figure 4.12.

4.4.5.2 Rebalancing

To restore the balance property, we *rebalance* the lowest node μ that is unbalanced, as follows:

- Let μ' be the child of μ whose subtree has maximum height, and μ'' be the child of μ' whose subtree has maximum height.
- Let (μ_1, μ_2, μ_3) be the left-to-right ordering of nodes $\{\mu, \mu', \mu''\}$, and (T_0, T_1, T_2, T_3) be the left-to-right ordering of the four subtrees of $\{\mu, \mu', \mu''\}$ not rooted at a node in $\{\mu, \mu', \mu''\}$.
- Replace the subtree rooted at μ with a new subtree rooted at μ_2, where μ_1 is the left child of μ_2 and has subtrees T_0 and T_1, and μ_3 is the right child of μ_2 and has subtrees T_2 and T_3.

Two examples of rebalancing are schematically shown in Figure 4.14. Other symmetric configurations are possible. In Figure 4.13 we show the rebalancing for the tree of Figure 4.12.

Note that the rebalancing causes all the nodes in the subtree of μ_2 to become balanced. Also, the subtree rooted at μ_2 now has the same height as the subtree rooted at node μ before insertion. This causes all of the previously unbalanced nodes to become balanced. To keep track of the nodes that become unbalanced, we can store at each node a *balance factor*, which is the difference of the heights of the left and right subtrees. A node becomes unbalanced when its balance factor becomes +2 or −2. It is easy to modify the algorithm for operation INSERT such that it maintains the balance factors of the nodes.

4.4.5.3 Deletion

The implementation of REMOVE for search trees given earlier in this section preserves the binary property, but may cause the balance property to be violated. After deleting a node, there can be only one unbalanced node, on the path from the deleted node to the root of T.

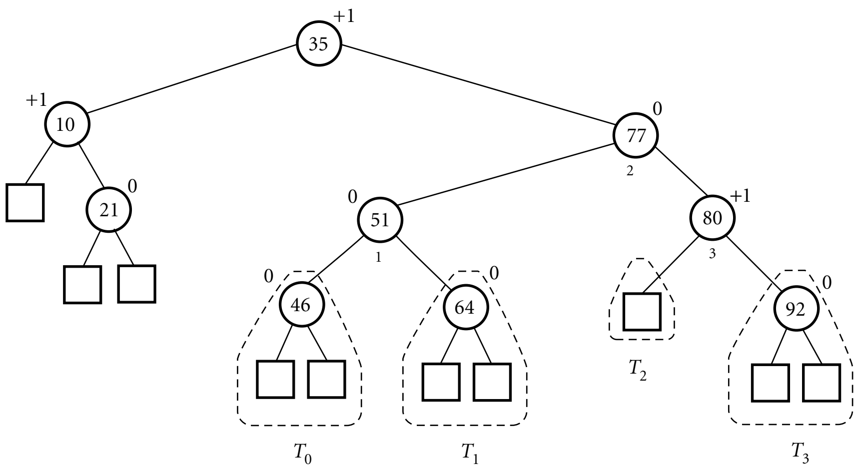


FIGURE 4.13 AVL-tree obtained by rebalancing the lowest unbalanced node in the tree of Figure 4.11. Note that all of the nodes are now balanced. The dashed lines identify the subtrees that participate in the rebalancing, as illustrated in Figure 4.14.

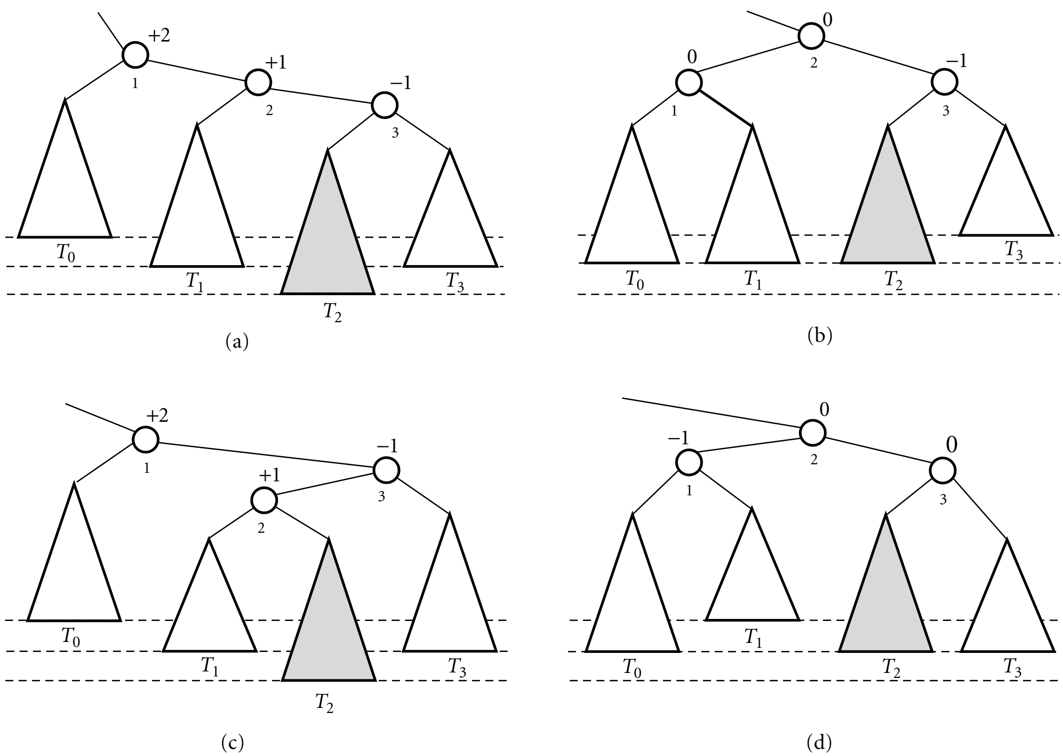


FIGURE 4.14 Schematic illustration of rebalancing a node in the INSERT algorithm for AVL-trees. The shaded subtree is the one where the new element was inserted. (a) and (b) Rebalancing by means of a single rotation. (c) and (d) Rebalancing by means of a double rotation.

TABLE 4.11 Performance of a Dictionary Realized by an AVL-Tree

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
NEXT	$O(\log N)$
PREV	$O(\log N)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
MODIFY	$O(\log N)$

To restore the balance property, we *rebalance* the unbalanced node using the previous algorithm, with minor modifications. If the subtrees of μ' have the same height, the height of the subtree rooted at μ_2 is the same as the height of the subtree rooted at μ before rebalancing, and we are done. If, instead, the subtrees of μ' do not have the same height, then the height of the subtree rooted at μ_2 is one less than the height of the subtree rooted at μ before rebalancing. This may cause an ancestor of μ_2 to become unbalanced, and we repeat the above computation. Balance factors are used to keep track of the nodes that become unbalanced, and can be easily maintained by the REMOVE algorithm.

4.4.5.4 Complexity

Let T be an AVL-tree storing N elements. The height of T is $O(\log N)$. Each dictionary operation affects only the nodes along a root-to-leaf path. Rebalancing a node takes $O(1)$ time. We conclude that each operation takes $O(\log N)$ time.

Table 4.11 shows the performance of a dictionary realized with an AVL-tree. In this table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.6 Realization with a Hash Table

The previous realizations of a dictionary make no assumptions on the structure of the keys and use comparisons between keys to guide the execution of the various operations.

4.4.6.1 Bucket Array

If the keys of a dictionary D are integers in the range $[1, M]$, we can implement D with a *bucket array* B. An element (x, y) of D is represented by setting $B[x] = y$. If an integer x is not in D, the location $B[x]$ stores a null value. In this implementation, we allocate a bucket for every possible element of D.

Table 4.12 shows the performance of a dictionary realized with a bucket array. In this table the keys in the dictionary are integers in the range $[1, M]$. The space complexity is $O(M)$.

The bucket array method can be extended to keys that are easily mapped to integers. For example, three-letter airport codes can be mapped to the integers in the range $[1, 26^3]$.

4.4.6.2 Hashing

The bucket array method works well when the range of keys is small. However, it is inefficient when the range of keys is large. To overcome this problem, we can use a *hash function* h that maps the keys of the original dictionary D into integers in the range $[1, M]$, where M is a parameter of the hash function. Now, we can apply the bucket array method using the *hashed value* $h(x)$ of the keys. In general, a *collision* may

TABLE 4.12 Performance of a Dictionary Realized by Bucket Array

Operation	Time
SIZE	$O(1)$
FIND	$O(1)$
LOCATEPREV	$O(M)$
LOCATENEXT	$O(M)$
NEXT	$O(M)$
PREV	$O(M)$
MIN	$O(M)$
MAX	$O(M)$
INSERT	$O(1)$
REMOVE	$O(1)$
MODIFY	$O(1)$

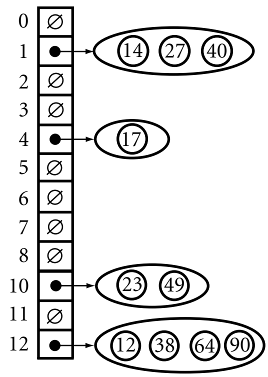


FIGURE 4.15 Example of a hash table of size 13 storing 10 elements. The hash function is $h(x) = x \bmod 13$.

happen, where two distinct keys x_1 and x_2 have the same hashed value, that is, $x_1 \neq x_2$ and $h(x_1) = h(x_2)$. Hence, each bucket must be able to accommodate a collection of elements.

A hash table of size M for a function $h(x)$ is a bucket array B of size M (primary structure) whose entries are dictionaries (secondary structures), such that element (x, y) is stored in the dictionary $B[h(x)]$. For simplicity of programming, the dictionaries used as secondary structures are typically realized with sequences. An example of a hash table is shown in Figure 4.15.

If all of the elements in the dictionary D collide, they are all stored in the same dictionary of the bucket array, and the performance of the hash table is the same as that of the kind of dictionary used for the secondary structures. At the other end of the spectrum, if no two elements of the dictionary D collide, they are stored in distinct one-element dictionaries of the bucket array, and the performance of the hash table is the same as that of a bucket array.

A typical hash function for integer keys is $h(x) = x \bmod M$ (here, the range is $[0, M-1]$). The size M of the hash table is usually chosen as a prime number. An example of a hash table is shown in Figure 4.15. It is interesting to analyze the performance of a hash table from a probabilistic viewpoint. If we assume that the hashed values of the keys are uniformly distributed in the range $[0, M-1]$, then each bucket holds on average N/M keys, where N is the size of the dictionary. Hence, when $N = O(M)$, the average size of the secondary data structures is $O(1)$.

Table 4.13 shows the performance of a dictionary realized with a hash table. Both the worst-case and average time complexity in the preceding probabilistic model are indicated. In this table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity

TABLE 4.13 Performance of a Dictionary Realized by a Hash Table of Size M

	Time	
Operation	Worst Case	Average
Size	$O(1)$	$O(1)$
Find	$O(N)$	$O(N/M)$
LocatePrev	$O(N+M)$	$O(N+M)$
LocateNext	$O(N+M)$	$O(N+M)$
Next	$O(N+M)$	$O(N+M)$
Prev	$O(N+M)$	$O(N+M)$
Min	$O(N+M)$	$O(N+M)$
Max	$O(N+M)$	$O(N+M)$
Insert	$O(1)$	$O(1)$
Remove	$O(1)$	$O(1)$
Modify	$O(1)$	$O(1)$

is $O(N+M)$. The average time complexity refers to a probabilistic model where the hashed values of the keys are uniformly distributed in the range $[1, M]$.

Acknowledgments

Work supported in part by the National Science Foundation under grant DUE–0231202. Bryan Cantrill contributed to this work while at Brown University.

Defining Terms

(a, b)-Tree: Search tree with additional properties (each node has between a and b children, and all the levels are full).

Abstract data type: Mathematically specified data type equipped with operations that can be performed on the objects.

AVL-tree: Binary search tree such that the subtrees of each node have heights that differ by at most one.

Binary search tree: Search tree such that each internal node has two children.

Bucket array: Implementation of a dictionary by means of an array indexed by the keys of the dictionary elements.

Container: Abstract data type storing a collection of objects (elements).

Dictionary: Container storing elements from a sorted universe supporting searches, insertions, and deletions.

Hash table: Implementation of a dictionary by means of a bucket array storing secondary dictionaries.

Heap: Binary tree with additional properties storing the elements of a priority queue.

Position: Object representing the place of an element stored in a container.

Locator: Mechanism for tracking an element stored in a container.

Priority queue: Container storing elements from a sorted universe that supports finding the maximum element, insertions, and deletions.

Search tree: Rooted ordered tree with additional properties storing the elements of a dictionary.

Sequence: Container storing objects in a linear order, supporting insertions (in a given position) and deletions.

References

Aggarwal, A. and Vitter, J.S. 1988. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127.

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. 1983. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

Chazelle, B. and Guibas, L.J. 1986. Fractional cascading. I. A data structuring technique. *Algorithmica*, 1:133–162.

Chiang, Y.-J. and Tamassia, R. 1992. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434.

Cohen, R.F. and Tamassia, R. 1995. Dynamic expression trees. *Algorithmica*, 13:245–265.

Comer, D. 1979. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. 2001. *Introduction to Algorithms.* MIT Press, Cambridge, MA.

Di Battista, G. and Tamassia, R. 1996. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318.

Di Battista, G., Eades, P., Tamassia, R., and Tollis, I.G. 1999. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ.

Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E. 1989. Making data structures persistent. *J. Comput. Syst. Sci.* 38:86–124.

Edelsbrunner, H. 1987. *Algorithms in Combinatorial Geometry,* Vol. 10, *EATCS Monographs on Theoretical Computer Science.* Springer–Verlag, Heidelberg, Germany.

Eppstein, D., Galil, Z., Italiano, G.F., and Nissenzweig, A. 1997. Sparsification: a technique for speeding up dynamic graph algorithms. *J. ACM*, 44:669–696.

Even, S. 1979. *Graph Algorithms*. Computer Science Press, Potomac, MD.

Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F. 1990. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA.

Frederickson, G.N. 1997. A data structure for dynamically maintaining rooted trees. *J. Algorithms*, 24:37–65.

Galil, Z. and Italiano, G.F. 1991. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344.

Gonnet, G.H. and Baeza-Yates, R. 1991. *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, MA.

Goodrich, M.T. and Tamassia, R. 2001. *Data Structures and Algorithms in Java*. Wiley, New York.

Hoffmann, K., Mehlhorn, K., Rosenstiehl, P., and Tarjan, R.E. 1986. Sorting Jordan sequences in linear time using level-linked search trees. *Inf. Control*, 68:170–184.

Horowitz, E., Sahni, S., and Metha, D. 1995. *Fundamentals of Data Structures in C++*. Computer Science Press, Potomac, MD.

Knuth, D.E. 1968. *Fundamental Algorithms.* Vol. I. In *The Art of Computer Programming*. Addison-Wesley, Reading, MA.

Knuth, D.E. 1973. *Sorting and Searching,* Vol. 3. In *The Art of Computer Programming.* Addison-Wesley, Reading, MA.

Mehlhorn, K. 1984. *Data Structures and Algorithms.* Vol. 1–3. Springer–Verlag.

Mehlhorn, K. and Näher, S. 1999. *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge University Press.

Mehlhorn, K. and Tsakalidis, A. 1990. Data structures. In *Algorithms and Complexity*. J. van Leeuwen, Ed., Vol. A, *Handbook of Theoretical Computer Science.* Elsevier, Amsterdam.

Munro, J.I. and Suwanda, H. 1980. Implicit Data Structures for Fast Search and Update. *J. Comput. Syst. Sci.*, 21:236–250.

Nievergelt, J. and Hinrichs, K.H. 1993. *Algorithms and Data Structures: With Applications to Graphics and Geometry.* Prentice Hall, Englewood Cliffs, NJ.

O'Rourke, J. 1994. *Computational Geometry in C.* Cambridge University Press,

Overmars, M.H. 1983. *The Design of Dynamic Data Structures,* Vol. 156, *Lecture Notes in Computer Science*. Springer-Verlag.

Preparata, F.P. and Shamos, M.I. 1985. *Computational Geometry: An Introduction.* Springer-Verlag, New York.

Pugh, W. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 35:668–676.

Sedgewick, R. 1992. *Algorithms in C++*. Addison-Wesley, Reading, MA.

Sleator, D.D. and Tarjan, R.E. 1993. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381.

Tamassia, R., Goodrich, M.T., Vismara, L., Handy, M., Shubina, G., Cohen R., Hudson, B., Baker, R.S., Gelfand, N., and Brandes, U. 2001. JDSL: the data structures library in Java. *Dr. Dobb's Journal*, 323:21–31.

Tarjan, R.E. 1983. *Data Structures and Network Algorithms, Vol. 44, CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial Applied Mathematics.

Vitter, J.S. and Flajolet, P. 1990. Average-case analysis of algorithms and data structures. In *Algorithms and Complexity,* J. van Leeuwen, Ed., Vol. A, *Handbook of Theoretical Computer Science,* pp. 431–524. Elsevier, Amsterdam.

Wood, D. 1993. *Data Structures, Algorithms, and Performance.* Addison-Wesley, Reading, MA.

Further Information

Many textbooks and monographs have been written on data structures, for example, Aho et al. [1983], Cormen et al. [2001], Gonnet and Baeza-Yates [1990], Goodrich and Tamassia [2001], Horowitz et al. [1995], Knuth [1968, 1973], Mehlhorn [1984], Nievergelt and Hinrichs [1993], Overmars [1983], Preparata and Shamos [1995], Sedgewick [1992], Tarjan [1983], and Wood [1993].

Papers surveying the state-of-the art in data structures include Chiang and Tamassia [1992], Galil and Italiano [1991], Mehlhorn and Tsakalidis [1990], and Vitter and Flajolet [1990].

JDSL is a library of fundamental data structures in Java [Tamassia et al. 2000]. LEDA is a library of advanced data structures in C++ [Mehlhorn and Näher 1999].

5

Complexity Theory

Eric W. Allender
Rutgers University

Michael C. Loui
University of Illinois at Urbana-Champaign

Kenneth W. Regan
State University of New York at Buffalo

5.1 Introduction

Computational complexity is the study of the difficulty of solving computational problems, in terms of the required computational resources, such as time and space (memory). Whereas the analysis of algorithms focuses on the time or space of an *individual* algorithm for a *specific* problem (such as sorting), complexity theory focuses on the **complexity class** of problems solvable in the same amount of time or space. Most common computational problems fall into a small number of complexity classes. Two important complexity classes are P, the set of problems that can be solved in polynomial time, and NP, the set of problems whose solutions can be verified in polynomial time.

By quantifying the resources required to solve a problem, complexity theory has profoundly affected our thinking about computation. Computability theory establishes the existence of undecidable problems, which cannot be solved in principle regardless of the amount of time invested. However, computability theory fails to find meaningful distinctions among decidable problems. In contrast, complexity theory establishes the existence of decidable problems that, although solvable in principle, cannot be solved in

practice because the time and space required would be larger than the age and size of the known universe [Stockmeyer and Chandra, 1979]. Thus, complexity theory characterizes the computationally feasible problems.

The quest for the boundaries of the set of feasible problems has led to the most important unsolved question in all of computer science: is P different from NP? Hundreds of fundamental problems, including many ubiquitous optimization problems of operations research, are **NP-complete**; they are the hardest problems in NP. If someone could find a polynomial-time algorithm for any one NP-complete problem, then there would be polynomial-time algorithms for all of them. Despite the concerted efforts of many scientists over several decades, no polynomial-time algorithm has been found for any NP-complete problem. Although we do not yet know whether P is different from NP, showing that a problem is NP-complete provides strong evidence that the problem is computationally infeasible and justifies the use of heuristics for solving the problem.

In this chapter, we define P, NP, and related complexity classes. We illustrate the use of **diagonalization** and **padding** techniques to prove relationships between classes. Next, we define NP-completeness, and we show how to prove that a problem is NP-complete. Finally, we define complexity classes for probabilistic and interactive computations.

Throughout this chapter, all numeric functions take integer arguments and produce integer values. All logarithms are taken to base 2. In particular, $\log n$ means $\lceil \log_2 n \rceil$.

5.2 Models of Computation

To develop a theory of the difficulty of computational problems, we need to specify precisely what a problem is, what an algorithm is, and what a measure of difficulty is. For simplicity, complexity theorists have chosen to represent problems as languages, to model algorithms by off-line multitape **Turing machines**, and to measure computational difficulty by the time and space required by a Turing machine. To justify these choices, some theorems of complexity theory show how to translate statements about, say, the time complexity of language recognition by Turing machines into statements about computational problems on more realistic models of computation. These theorems imply that the principles of complexity theory are not artifacts of Turing machines, but intrinsic properties of computation.

This section defines different kinds of Turing machines. The deterministic Turing machine models actual computers. The nondeterministic Turing machine is not a realistic model, but it helps classify the complexity of important computational problems. The alternating Turing machine models a form of parallel computation, and it helps elucidate the relationship between time and space.

5.2.1 Computational Problems and Languages

Computer scientists have invented many elegant formalisms for representing data and control structures. Fundamentally, all representations are patterns of symbols. Therefore, we represent an instance of a computational problem as a sequence of symbols.

Let Σ be a finite set, called the *alphabet*. A *word* over Σ is a finite sequence of symbols from Σ. Sometimes a word is called a *string*. Let Σ^* denote the set of all words over Σ. For example, if $\Sigma = \{\mathbf{0}, \mathbf{1}\}$, then

$$\Sigma^* = \{\lambda, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{01}, \mathbf{10}, \mathbf{11}, \mathbf{000}, \ldots\}$$

is the set of all binary words, including the empty word λ. The *length* of a word w, denoted by $|w|$, is the number of symbols in w. A *language* over Σ is a subset of Σ^*.

A *decision problem* is a computational problem whose answer is simply `yes` or `no`. For example, is the input graph connected, or is the input a sorted list of integers? A decision problem can be expressed as a membership problem for a language A: for an input x, does x belong to A? For a language A that represents connected graphs, the input word x might represent an input graph G, and $x \in A$ if and only if G is connected.

For every decision problem, the representation should allow for easy parsing, to determine whether a word represents a legitimate instance of the problem. Furthermore, the representation should be concise. In particular, it would be unfair to encode the answer to the problem into the representation of an instance of the problem; for example, for the problem of deciding whether an input graph is connected, the representation should not have an extra bit that tells whether the graph is connected. A set of integers $S = \{x_1, \dots, x_m\}$ is represented by listing the binary representation of each x_i, with the representations of consecutive integers in S separated by a nonbinary symbol. A graph is naturally represented by giving either its adjacency matrix or a set of adjacency lists, where the list for each vertex v specifies the vertices adjacent to v.

Whereas the solution to a decision problem is `yes` or `no`, the solution to an optimization problem is more complicated; for example, determine the shortest path from vertex u to vertex v in an input graph G. Nevertheless, for every optimization (minimization) problem, with objective function g, there is a corresponding decision problem that asks whether there exists a feasible solution z such that $g(z) \leq k$, where k is a given target value. Clearly, if there is an algorithm that solves an optimization problem, then that algorithm can be used to solve the corresponding decision problem. Conversely, if an algorithm solves the decision problem, then with a binary search on the range of values of g, we can determine the optimal value. Moreover, using a decision problem as a subroutine often enables us to construct an optimal solution; for example, if we are trying to find a shortest path, we can use a decision problem that determines if a shortest path starting from a given vertex uses a given edge. Therefore, there is little loss of generality in considering only decision problems, represented as language membership problems.

5.2.2 Turing Machines

This subsection and the next three give precise, formal definitions of Turing machines and their variants. These subsections are intended for reference. For the rest of this chapter, the reader need not understand these definitions in detail, but may generally substitute "program" or "computer" for each reference to "Turing machine."

A k-worktape **Turing machine** M consists of the following:

- A finite set of states Q, with special states q_0 (initial state), q_A (accept state), and q_R (reject state).
- A finite alphabet Σ, and a special blank symbol $\square \notin \Sigma$.
- The $k + 1$ linear tapes, each divided into cells. Tape 0 is the *input tape*, and tapes $1, \dots, k$ are the *worktapes*. Each tape is infinite to the left and to the right. Each cell holds a single symbol from $\Sigma \cup \{\square\}$. By convention, the input tape is read only. Each tape has an access head, and at every instant, each access head scans one cell (see Figure 5.1).

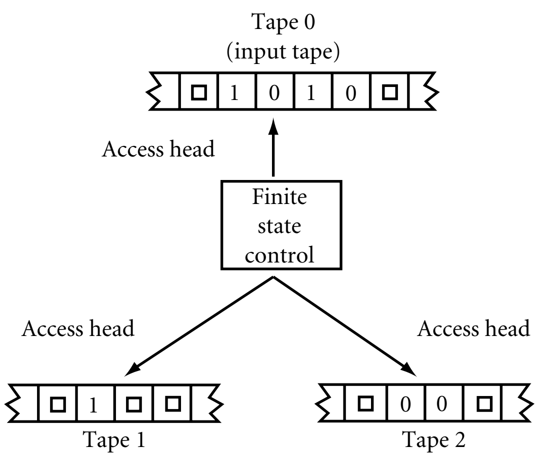


FIGURE 5.1 A two-tape Turing machine.

- A finite transition table δ, which comprises tuples of the form

$$(q, s_0, s_1, \dots, s_k, q', s'_1, \dots, s'_k, d_0, d_1, \dots, d_k)$$

  where $q, q' \in Q$, each $s_i, s'_i \in \Sigma \cup \{\square\}$, and each $d_i \in \{-1, 0, +1\}$.

  A tuple specifies a step of M: if the current state is q, and $s_0, s_1, \dots, s_k$ are the symbols in the cells scanned by the access heads, then M replaces s_i by s'_i for $i = 1, \dots, k$ simultaneously, changes state to q', and moves the head on tape i one cell to the left ($d_i = -1$) or right ($d_i = +1$) or not at all ($d_i = 0$) for $i = 0, \dots, k$. Note that M cannot write on tape 0, that is, M can write only on the worktapes, not on the input tape.
- In a tuple, no s'_i can be the blank symbol $\square$. Because M may not write a blank, the worktape cells that its access heads previously visited are nonblank.
- No tuple contains q_A or q_R as its first component. Thus, once M enters state q_A or state q_R, it stops.
- Initially, M is in state q_0, an input word in Σ^* is inscribed on contiguous cells of the input tape, the access head on the input tape is on the leftmost symbol of the input word, and all other cells of all tapes contain the blank symbol $\square$.

The Turing machine M that we have defined is *nondeterministic:* δ may have several tuples with the same combination of state q and symbols $s_0, s_1, \dots, s_k$ as the first $k+2$ components, so that M may have several possible next steps. A machine M is *deterministic* if for every combination of state q and symbols $s_0, s_1, \dots, s_k$, at most one tuple in δ contains the combination as its first $k+2$ components. A deterministic machine always has at most one possible next step.

A *configuration* of a Turing machine M specifies the current state, the contents of all tapes, and the positions of all access heads.

A *computation path* is a sequence of configurations $C_0, C_1, \dots, C_t, \dots$, where C_0 is the initial configuration of M, and each C_{j+1} follows from C_j in one step by applying the changes specified by a tuple in δ. If no tuple is applicable to C_t, then C_t is *terminal*, and the computation path is *halting*. If M has no infinite computation paths, then M *always halts.*

A halting computation path is *accepting* if the state in the last configuration C_t is q_A; otherwise it is *rejecting*. By adding tuples to the program if needed, we can ensure that every rejecting computation ends in state q_R. This leaves the question of computation paths that do not halt. In complexity theory, we rule this out by considering only machines whose computation paths *always halt*. M *accepts* an input word x if there exists an accepting computation path that starts from the initial configuration in which x is on the input tape. For nondeterministic M, it does not matter if some other computation paths end at q_R. If M is deterministic, then there is at most one halting computation path, hence at most one accepting path.

The *language accepted by* M, written $L(M)$, is the set of words accepted by M. If $A = L(M)$, and M always halts, then M *decides* A.

In addition to deciding languages, deterministic Turing machines can compute functions. Designate tape 1 to be the *output tape*. If M halts on input word x, then the nonblank word on tape 1 in the final configuration is the output of M. A function f is *total recursive* if there exists a deterministic Turing machine M that always halts such that for each input word x, the output of M is the value of $f(x)$.

Almost all results in complexity theory are insensitive to minor variations in the underlying computational models. For example, we could have chosen Turing machines whose tapes are restricted to be only one-way infinite or whose alphabet is restricted to $\{\mathbf{0}, \mathbf{1}\}$. It is straightforward to simulate a Turing machine as defined by one of these restricted Turing machines, one step at a time: each step of the original machine can be simulated by $O(1)$ steps of the restricted machine.

5.2.3 Universal Turing Machines

Chapter 6 states that there exists a *universal Turing machine* U, which takes as input a string $\langle M, x\rangle$ that encodes a Turing machine M and a word x, and simulates the operation of M on x, and U accepts $\langle M, x\rangle$ if and only if M accepts x. A theorem of Hennie and Stearns [1966] implies that the machine U can be

constructed to have only two worktapes, such that U can simulate any t steps of M in only $O(t \log t)$ steps of its own, using only $O(1)$ times the worktape cells used by M. The constants implicit in these big-O bounds may depend on M.

We can think of U with a fixed M as a machine U_M and define $L(U_M) = \{x : U \text{ accepts } \langle M, x \rangle\}$. Then $L(U_M) = L(M)$. If M always halts, then U_M always halts; and if M is deterministic, then U_M is deterministic.

5.2.4 Alternating Turing Machines

By definition, a nondeterministic Turing machine M accepts its input word x if there exists an accepting computation path, starting from the initial configuration with x on the input tape. Let us call a *configuration* C accepting if there is a computation path of M that starts in C and ends in a configuration whose state is q_A. Equivalently, a configuration C is accepting if either the state in C is q_A or there exists an accepting configuration C' reachable from C by one step of M. Then M accepts x if the initial configuration with input word x is accepting.

The *alternating Turing machine* generalizes this notion of acceptance. In an alternating Turing machine M, each state is labeled either existential or universal. (Do not confuse the universal state in an alternating Turing machine with the universal Turing machine.) A nonterminal configuration C is existential (respectively, universal) if the state in C is labeled existential (universal). A terminal configuration is accepting if its state is q_A. A nonterminal existential configuration C is accepting if there *exists* an accepting configuration C' reachable from C by one step of M. A nonterminal universal configuration C is accepting if for *every* configuration C' reachable from C by one step of M, the configuration C' is accepting. Finally, M accepts x if the initial configuration with input word x is an accepting configuration.

A nondeterministic Turing machine is thus a special case of an alternating Turing machine in which every state is existential.

The computation of an alternating Turing machine M alternates between existential states and universal states. Intuitively, from an existential configuration, M guesses a step that leads toward acceptance; from a universal configuration, M checks whether each possible next step leads toward acceptance — in a sense, M checks all possible choices in parallel. An alternating computation captures the essence of a two-player game: player 1 has a winning strategy if there exists a move for player 1 such that for every move by player 2, there exists a subsequent move by player 1, etc., such that player 1 eventually wins.

5.2.5 Oracle Turing Machines

Some computational problems remain difficult even when solutions to instances of a particular, different decision problem are available for free. When we study the complexity of a problem *relative* to a language A, we assume that answers about membership in A have been precomputed and stored in a (possibly infinite) table and that there is no cost to obtain an answer to a membership query: Is w in A? The language A is called an **oracle**. Conceptually, an algorithm queries the oracle whether a word w is in A, and it receives the correct answer in one step.

An *oracle Turing machine* is a Turing machine M with a special *oracle tape* and special states QUERY, YES, and NO. The computation of the oracle Turing machine M^A, with oracle language A, is the same as that of an ordinary Turing machine, except that when M enters the QUERY state with a word w on the oracle tape, in one step, M enters either the YES state if $w \in A$ or the NO state if $w \notin A$. Furthermore, during this step, the oracle tape is erased, so that the time for setting up each query is accounted for separately.

5.3 Resources and Complexity Classes

In this section, we define the measures of difficulty of solving computational problems. We introduce complexity classes, which enable us to classify problems according to the difficulty of their solution.

5.3.1 Time and Space

We measure the difficulty of a computational problem by the running time and the space (memory) requirements of an algorithm that solves the problem. Clearly, in general, a finite algorithm cannot have a table of all answers to infinitely many instances of the problem, although an algorithm could look up precomputed answers to a finite number of instances; in terms of Turing machines, the finite answer table is built into the set of states and the transition table. For these instances, the running time is negligible — just the time needed to read the input word. Consequently, our complexity measure should consider a whole problem, not only specific instances.

We express the complexity of a problem, in terms of the growth of the required time or space, as a function of the length n of the input word that encodes a problem instance. We consider the worst-case complexity, that is, for each n, the maximum time or space required among all inputs of length n.

Let M be a Turing machine that always halts. The *time* taken by M on input word x, denoted by $\mathrm{Time}_M(x)$, is defined as follows:

- If M accepts x, then $\mathrm{Time}_M(x)$ is the number of steps in the shortest accepting computation path for x.
- If M rejects x, then $\mathrm{Time}_M(x)$ is the number of steps in the longest computation path for x.

For a deterministic machine M, for every input x, there is at most one halting computation path, and its length is $\mathrm{Time}_M(x)$. For a nondeterministic machine M, if $x \in L(M)$, then M can guess the correct steps to take toward an accepting configuration, and $\mathrm{Time}_M(x)$ measures the length of the path on which M always makes the best guess.

The *space* used by a Turing machine M on input x, denoted by $\mathrm{Space}_M(x)$, is defined as follows. The space used by a halting computation path is the number of nonblank worktape cells in the last configuration; this is the number of different cells ever written by the worktape heads of M during the computation path, since M never writes the blank symbol. Because the space occupied by the input word is not counted, a machine can use a sublinear ($o(n)$) amount of space.

- If M accepts x, then $\mathrm{Space}_M(x)$ is the minimum space used among all accepting computation paths for x.
- If M rejects x, then $\mathrm{Space}_M(x)$ is the maximum space used among all computation paths for x.

The **time complexity** of a machine M is the function

$$t(n) = \max\{\mathrm{Time}_M(x) : |x| = n\}$$

We assume that M reads all of its input word, and the blank symbol after the right end of the input word, so $t(n) \geq n + 1$. The **space complexity** of M is the function

$$s(n) = \max\{\mathrm{Space}_M(x) : |x| = n\}$$

Because few interesting languages can be decided by machines of sublogarithmic space complexity, we henceforth assume that $s(n) \geq \log n$.

A function $f(x)$ is *computable in polynomial time* if there exists a deterministic Turing machine M of polynomial time complexity such that for each input word x, the output of M is $f(x)$.

5.3.2 Complexity Classes

Having defined the time complexity and space complexity of individual Turing machines, we now define classes of languages with particular complexity bounds. These definitions will lead to definitions of P and NP.

Let $t(n)$ and $s(n)$ be numeric functions. Define the following classes of languages:

- $\mathsf{DTIME}[t(n)]$ is the class of languages decided by deterministic Turing machines of time complexity $O(t(n))$.
- $\mathsf{NTIME}[t(n)]$ is the class of languages decided by nondeterministic Turing machines of time complexity $O(t(n))$.
- $\mathsf{DSPACE}[s(n)]$ is the class of languages decided by deterministic Turing machines of space complexity $O(s(n))$.
- $\mathsf{NSPACE}[s(n)]$ is the class of languages decided by nondeterministic Turing machines of space complexity $O(s(n))$.

We sometimes abbreviate $\mathsf{DTIME}[t(n)]$ to $\mathsf{DTIME}[t]$ (and so on) when t is understood to be a function, and when no reference is made to the input length n.

The following are the **canonical complexity classes**:

- $\mathsf{L} = \mathsf{DSPACE}[\log n]$ (deterministic log space)
- $\mathsf{NL} = \mathsf{NSPACE}[\log n]$ (nondeterministic log space)
- $\mathsf{P} = \mathsf{DTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[n^k]$ (polynomial time)
- $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[n^k]$ (nondeterministic polynomial time)
- $\mathsf{PSPACE} = \mathsf{DSPACE}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[n^k]$ (polynomial space)
- $\mathsf{E} = \mathsf{DTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[k^n]$
- $\mathsf{NE} = \mathsf{NTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[k^n]$
- $\mathsf{EXP} = \mathsf{DTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{DTIME}[2^{n^k}]$ (deterministic exponential time)
- $\mathsf{NEXP} = \mathsf{NTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{NTIME}[2^{n^k}]$ (nondeterministic exponential time)

The space classes L and PSPACE are defined in terms of the DSPACE complexity measure. By Savitch's Theorem (see Theorem 5.2), the NSPACE measure with polynomial bounds also yields PSPACE.

The class P contains many familiar problems that can be solved efficiently, such as (decision problem versions of) finding shortest paths in networks, parsing for context-free languages, sorting, matrix multiplication, and linear programming. Consequently, P has become accepted as representing the set of computationally feasible problems. Although one could legitimately argue that a problem whose best algorithm has time complexity $\Theta(n^{99})$ is really infeasible, in practice, the time complexities of the vast majority of known polynomial-time algorithms have low degrees: they run in $O(n^4)$ time or less. Moreover, P is a robust class: although defined by Turing machines, P remains the same when defined by other models of sequential computation. For example, random access machines (RAMs) (a more realistic model of computation defined in Chapter 6) can be used to define P because Turing machines and RAMs can simulate each other with polynomial-time overhead.

The class NP can also be defined by means other than nondeterministic Turing machines. NP equals the class of problems whose solutions can be *verified* quickly, by deterministic machines in polynomial time. Equivalently, NP comprises those languages whose membership proofs can be checked quickly.

For example, one language in NP is the set of satisfiable Boolean formulas, called SAT. A Boolean formula ϕ is satisfiable if there exists a way of assigning `true` or `false` to each variable such that under this truth assignment, the value of ϕ is `true`. For example, the formula $x \wedge (\overline{x} \vee y)$ is satisfiable, but $x \wedge \overline{y} \wedge (\overline{x} \vee y)$ is not satisfiable. A nondeterministic Turing machine M, after checking the syntax of ϕ and counting the number n of variables, can nondeterministically write down an n-bit 0-1 string a on its tape, and then deterministically (and easily) evaluate ϕ for the truth assignment denoted by a. The computation path corresponding to each individual a accepts if and only if $\phi(a) = \texttt{true}$, and so M itself accepts ϕ if and only if ϕ is satisfiable; that is, $L(M) = \text{SAT}$. Again, this checking of given assignments differs significantly from trying to *find* an accepting assignment.

Another language in NP is the set of undirected graphs with a *Hamiltonian circuit*, that is, a path of edges that visits each vertex exactly once and returns to the starting point. If a solution exists and is given, its

correctness can be verified quickly. Finding such a circuit, however, or proving one does not exist, appears to be computationally difficult.

The characterization of NP as the set of problems with easily verified solutions is formalized as follows: $A \in \mathsf{NP}$ if and only if there exist a language $A' \in \mathsf{P}$ and a polynomial p such that for every x, $x \in A$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in A'$. Here, whenever x belongs to A, y is interpreted as a positive solution to the problem represented by x, or equivalently, as a proof that x belongs to A. The difference between P and NP is that between solving and checking, or between finding a proof of a mathematical theorem and testing whether a candidate proof is correct. In essence, NP represents all sets of theorems with proofs that are short (i.e., of polynomial length) and checkable quickly (i.e., in polynomial time), while P represents those statements that can proved or refuted quickly from scratch.

Further motivation for studying L, NL, and PSPACE comes from their relationships to P and NP. Namely, L and NL are the largest space-bounded classes known to be contained in P, and PSPACE is the smallest space-bounded class known to contain NP. (It is worth mentioning here that NP does not stand for "non-polynomial time"; the class P is a subclass of NP.) Similarly, EXP is of interest primarily because it is the smallest deterministic time class known to contain NP. The closely related class E is not known to contain NP.

5.4 Relationships between Complexity Classes

The P versus NP question asks about the relationship between these complexity classes: Is P a proper subset of NP, or does $\mathsf{P} = \mathsf{NP}$? Much of complexity theory focuses on the relationships between complexity classes because these relationships have implications for the difficulty of solving computational problems. In this section, we summarize important known relationships. We demonstrate two techniques for proving relationships between classes: diagonalization and padding.

5.4.1 Constructibility

The most basic theorem that one should expect from complexity theory would say, "If you have more resources, you can do more." Unfortunately, if we are not careful with our definitions, then this claim is false:

Theorem 5.1 (Gap Theorem) *There is a computable, strictly increasing time bound $t(n)$ such that* $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[2^{2^{t(n)}}]$ [Borodin, 1972].

That is, there is an empty gap between time $t(n)$ and time doubly-exponentially greater than $t(n)$, in the sense that anything that can be computed in the larger time bound can already be computed in the smaller time bound. That is, even with much more time, you can not compute more. This gap can be made much larger than doubly-exponential; for any computable r, there is a computable time bound t such that $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[r(t(n))]$. Exactly analogous statements hold for the NTIME, DSPACE, and NSPACE measures.

Fortunately, the gap phenomenon cannot happen for time bounds t that anyone would ever be interested in. Indeed, the proof of the Gap Theorem proceeds by showing that one can define a time bound t such that no machine has a running time that is between $t(n)$ and $2^{2^{t(n)}}$. This theorem indicates the need for formulating only those time bounds that actually describe the complexity of some machine.

A function $t(n)$ is **time-constructible** if there exists a deterministic Turing machine that halts after exactly $t(n)$ steps for every input of length n. A function $s(n)$ is **space-constructible** if there exists a deterministic Turing machine that uses exactly $s(n)$ worktape cells for every input of length n. (Most authors consider only functions $t(n) \geq n + 1$ to be time-constructible, and many limit attention to $s(n) \geq \log n$ for space bounds. There do exist sub-logarithmic space-constructible functions, but we prefer to avoid the tricky theory of $o(\log n)$ space bounds.)

For example, $t(n) = n + 1$ is time-constructible. Furthermore, if $t_1(n)$ and $t_2(n)$ are time-constructible, then so are the functions $t_1 + t_2$, $t_1 t_2$, $t_1^{t_2}$, and c^{t_1} for every integer $c > 1$. Consequently, if $p(n)$ is a polynomial, then $p(n) = \Theta(t(n))$ for some time-constructible polynomial function $t(n)$. Similarly, $s(n) = \log n$ is space-constructible, and if $s_1(n)$ and $s_2(n)$ are space-constructible, then so are the functions $s_1 + s_2$, $s_1 s_2$, $s_1^{s_2}$, and c^{s_1} for every integer $c > 1$. Many common functions are space-constructible: for example, $n \log n$, n^3, 2^n, $n!$.

Constructibility helps eliminate an arbitrary choice in the definition of the basic time and space classes. For general time functions t, the classes $\mathsf{DTIME}[t]$ and $\mathsf{NTIME}[t]$ may vary depending on whether machines are required to halt within t steps on all computation paths, or just on those paths that accept. If t is time-constructible and s is space-constructible, however, then $\mathsf{DTIME}[t]$, $\mathsf{NTIME}[t]$, $\mathsf{DSPACE}[s]$, and $\mathsf{NSPACE}[s]$ can be defined without loss of generality in terms of Turing machines that always halt.

As a general rule, any function $t(n) \geq n + 1$ and any function $s(n) \geq \log n$ that one is interested in as a time or space bound, is time- or space-constructible, respectively. As we have seen, little of interest can be proved without restricting attention to constructible functions. This restriction still leaves a rich class of resource bounds.

5.4.2 Basic Relationships

Clearly, for all time functions $t(n)$ and space functions $s(n)$, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{NTIME}[t(n)]$ and $\mathsf{DSPACE}[s(n)] \subseteq \mathsf{NSPACE}[s(n)]$ because a deterministic machine is a special case of a nondeterministic machine. Furthermore, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$ and $\mathsf{NTIME}[t(n)] \subseteq \mathsf{NSPACE}[t(n)]$ because at each step, a k-tape Turing machine can write on at most $k = O(1)$ previously unwritten cells. The next theorem presents additional important relationships between classes.

Theorem 5.2 *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DTIME}[2^{O(t(n))}]$
(b) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$
(c) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$
(d) **(Savitch's Theorem)** $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$ [Savitch, 1970]

As a consequence of the first part of this theorem, $\mathsf{NP} \subseteq \mathsf{EXP}$. No better general upper bound on deterministic time is known for languages in NP, however. See Figure 5.2 for other known inclusion relationships between canonical complexity classes.

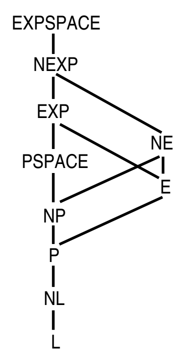


FIGURE 5.2 Inclusion relationships between the canonical complexity classes.

Although we do not know whether allowing nondeterminism strictly increases the class of languages decided in polynomial time, Savitch's Theorem says that for space classes, nondeterminism does not help by more than a polynomial amount.

5.4.3 Complementation

For a language A over an alphabet Σ, define $\overline{A}$ to be the complement of A in the set of words over Σ: that is, $\overline{A} = \Sigma^* - A$. For a class of languages $\mathcal{C}$, define co-$\mathcal{C} = \{\, \overline{A} : A \in \mathcal{C} \,\}$. If $\mathcal{C} = $ co-$\mathcal{C}$, then $\mathcal{C}$ is **closed under complementation.**

In particular, co-NP is the class of languages that are complements of languages in NP. For the language SAT of satisfiable Boolean formulas, $\overline{\text{SAT}}$ is essentially the set of unsatisfiable formulas, whose value is `false` for every truth assignment, together with the syntactically incorrect formulas. A closely related language in co-NP is the set of Boolean tautologies, namely, those formulas whose value is `true` for every truth assignment. The question of whether NP equals co-NP comes down to whether every tautology has a short (i.e., polynomial-sized) proof. The only obvious general way to prove a tautology ϕ in m variables is to verify all 2^m rows of the truth table for ϕ, taking exponential time. Most complexity theorists believe that there is no general way to reduce this time to polynomial, hence that NP $\neq$ co-NP.

Questions about complementation bear directly on the P vs. NP question. It is easy to show that P is closed under complementation (see the next theorem). Consequently, if NP $\neq$ co-NP, then P $\neq$ NP.

Theorem 5.3 (Complementation Theorems) *Let t be a time-constructible function, and let s be a space-constructible function, with $s(n) \geq \log n$ for all n. Then,*

1. $\mathsf{DTIME}[t]$ *is closed under complementation.*
2. $\mathsf{DSPACE}[s]$ *is closed under complementation.*
3. $\mathsf{NSPACE}[s]$ *is closed under complementation* [Immerman, 1988; Szelepcsényi, 1988].

The Complementation Theorems are used to prove the Hierarchy Theorems in the next section.

5.4.4 Hierarchy Theorems and Diagonalization

A hierarchy theorem is a theorem that says, "If you have more resources, you can compute more." As we saw in Section 5.4.1, this theorem is possible only if we restrict attention to constructible time and space bounds. Next, we state hierarchy theorems for deterministic and nondeterministic time and space classes. In the following, $\subset$ denotes *strict* inclusion between complexity classes.

Theorem 5.4 (Hierarchy Theorems) *Let t_1 and t_2 be time-constructible functions, and let s_1 and s_2 be space-constructible functions, with $s_1(n), s_2(n) \geq \log n$ for all n.*

(a) *If $t_1(n) \log t_1(n) = o(t_2(n))$, then* $\mathsf{DTIME}[t_1] \subset \mathsf{DTIME}[t_2]$.
(b) *If $t_1(n+1) = o(t_2(n))$, then* $\mathsf{NTIME}[t_1] \subset \mathsf{NTIME}[t_2]$ [Seiferas et al., 1978].
(c) *If $s_1(n) = o(s_2(n))$, then* $\mathsf{DSPACE}[s_1] \subset \mathsf{DSPACE}[s_2]$.
(d) *If $s_1(n) = o(s_2(n))$, then* $\mathsf{NSPACE}[s_1] \subset \mathsf{NSPACE}[s_2]$.

As a corollary of the Hierarchy Theorem for DTIME,

$$\mathsf{P} \subseteq \mathsf{DTIME}[n^{\log n}] \subset \mathsf{DTIME}[2^n] \subseteq \mathsf{E};$$

hence, we have the strict inclusion P $\subset$ E. Although we do not know whether P $\subset$ NP, there exists a problem in E that cannot be solved in polynomial time. Other consequences of the Hierarchy Theorems are NE $\subset$ NEXP and NL $\subset$ PSPACE.

In the Hierarchy Theorem for DTIME, the hypothesis on t_1 and t_2 is $t_1(n) \log t_1(n) = o(t_2(n))$, instead of $t_1(n) = o(t_2(n))$, for technical reasons related to the simulation of machines with multiple worktapes by a single universal Turing machine with a fixed number of worktapes. Other computational models, such as random access machines, enjoy tighter time hierarchy theorems.

All proofs of the Hierarchy Theorems use the technique of **diagonalization**. For example, the proof for DTIME constructs a Turing machine M of time complexity t_2 that considers all machines $M_1, M_2, \ldots$ whose time complexity is t_1; for each i, the proof finds a word x_i that is accepted by M if and only if $x_i \notin L(M_i)$, the language decided by M_i. Consequently, $L(M)$, the language decided by M, differs from each $L(M_i)$, hence $L(M) \notin \mathsf{DTIME}[t_1]$. The diagonalization technique resembles the classic method used to prove that the real numbers are uncountable, by constructing a number whose j^{th} digit differs from the j^{th} digit of the j^{th} number on the list. To illustrate the diagonalization technique, we outline the proof of the Hierarchy Theorem for DSPACE. In this subsection, $\langle i, x\rangle$ stands for the string $0^i 1x$, and *zeroes*(y) stands for the number of 0's that a given string y starts with. Note that *zeroes*$(\langle i, x\rangle) = i$.

Proof (of the DSPACE Hierarchy Theorem)
We construct a deterministic Turing machine M that decides a language A such that $A \in \mathsf{DSPACE}[s_2] - \mathsf{DSPACE}[s_1]$.

Let U be a deterministic universal Turing machine, as described in Section 5.2.3. On input x of length n, machine M performs the following:

1. Lay out $s_2(n)$ cells on a worktape.
2. Let $i =$ *zeroes*(x).
3. Simulate the universal machine U on input $\langle i, x\rangle$. Accept x if U tries to use more than s_2 worktape cells. (We omit some technical details, and the way in which the constructibility of s_2 is used to ensure that this process halts.)
4. If U accepts $\langle i, x\rangle$, then reject; if U rejects $\langle i, x\rangle$, then accept.

Clearly, M always halts and uses space $O(s_2(n))$. Let $A = L(M)$.

Suppose $A \in \mathsf{DSPACE}[s_1(n)]$. Then there is some Turing machine M_j accepting A using space at most $s_1(n)$. Since the space used by U is $O(1)$ times the space used by M_j, there is a constant k depending only on j (in fact, we can take $k = |j|$), such that U, on inputs z of the form $z = \langle j, x\rangle$, uses at most $ks_1(|x|)$ space.

Since $s_1(n) = o(s_2(n))$, there is an n_0 such that $ks_1(n) \leq s_2(n)$ for all $n \geq n_0$. Let x be a string of length greater than n_0 such that the first $j+1$ symbols of x are $0^j 1$. Note that the universal Turing machine U, on input $\langle j, x\rangle$, simulates M_j on input x and uses space at most $ks_1(n) \leq s_2(n)$. Thus, when we consider the machine M defining A, we see that on input x the simulation does not stop in step 3, but continues on to step 4, and thus $x \in A$ if and only if U rejects $\langle j, x\rangle$. Consequently, M_j does not accept A, contrary to our assumption. Thus, $A \notin \mathsf{DSPACE}[s_1(n)]$. □

Although the diagonalization technique successfully separates some pairs of complexity classes, diagonalization does not seem strong enough to separate P from NP. (See Theorem 5.10 below.)

5.4.5 Padding Arguments

A useful technique for establishing relationships between complexity classes is the **padding argument.** Let A be a language over alphabet Σ, and let # be a symbol not in Σ. Let f be a numeric function. The ***f*-padded version of** L is the language

$$A' = \{x\#^{f(n)} : x \in A \text{ and } n = |x|\}.$$

That is, each word of A' is a word in A concatenated with $f(n)$ consecutive # symbols. The padded version A' has the same information content as A, but because each word is longer, the computational complexity of A' is smaller.

The proof of the next theorem illustrates the use of a padding argument.

Theorem 5.5 *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{E} = \mathsf{NE}$ [Book, 1974].

Proof Since $\mathsf{E} \subseteq \mathsf{NE}$, we prove that $\mathsf{NE} \subseteq \mathsf{E}$.

Let $A \in \mathsf{NE}$ be decided by a nondeterministic Turing machine M in at most $t(n) = k^n$ time for some constant integer k. Let A' be the $t(n)$-padded version of A. From M, we construct a nondeterministic Turing machine M' that decides A' in linear time: M' checks that its input has the correct format, using the time-constructibility of t; then M' runs M on the prefix of the input preceding the first # symbol. Thus, $A' \in \mathsf{NP}$.

If $\mathsf{P} = \mathsf{NP}$, then there is a deterministic Turing machine D' that decides A' in at most $p'(n)$ time for some polynomial p'. From D', we construct a deterministic Turing machine D that decides A, as follows. On input x of length n, since $t(n)$ is time-constructible, machine D constructs $x\#^{t(n)}$, whose length is $n + t(n)$, in $O(t(n))$ time. Then D runs D' on this input word. The time complexity of D is at most $O(t(n)) + p'(n + t(n)) = 2^{O(n)}$. Therefore, $\mathsf{NE} \subseteq \mathsf{E}$. □

A similar argument shows that the $\mathsf{E} = \mathsf{NE}$ question is equivalent to the question of whether $\mathsf{NP} - \mathsf{P}$ contains a subset of 1^*, that is, a language over a single-letter alphabet.

5.5 Reducibility and Completeness

In this section, we discuss relationships between problems: informally, if one problem reduces to another problem, then in a sense, the second problem is harder than the first. The hardest problems in NP are the NP-complete problems. We define NP-completeness precisely, and we show how to prove that a problem is NP-complete. The theory of NP-completeness, together with the many known NP-complete problems, is perhaps the best justification for interest in the classes P and NP. All of the other canonical complexity classes listed above have natural and important problems that are complete for them; we give some of these as well.

5.5.1 Resource-Bounded Reducibilities

In mathematics, as in everyday life, a typical way to solve a new problem is to reduce it to a previously solved problem. Frequently, an instance of the new problem is expressed completely in terms of an instance of the prior problem, and the solution is then interpreted in the terms of the new problem. For example, the maximum weighted matching problem for bipartite graphs (also called the assignment problem) reduces to the network flow problem (see Chapter 7). This kind of reduction is called **many-one reducibility**, and is defined below.

A different way to solve the new problem is to use a subroutine that solves the prior problem. For example, we can solve an optimization problem whose solution is feasible and maximizes the value of an objective function g by repeatedly calling a subroutine that solves the corresponding decision problem of whether there exists a feasible solution x whose value $g(x)$ satisfies $g(x) \geq k$. This kind of reduction is called **Turing reducibility**, and is also defined below.

Let A_1 and A_2 be languages. A_1 is many-one reducible to A_2, written $A_1 \leq_m A_2$, if there exists a total recursive function f such that for all x, $x \in A_1$ if and only if $f(x) \in A_2$. The function f is called the **transformation function.** A_1 is Turing reducible to A_2, written $A_1 \leq_T A_2$, if A_1 can be decided by a deterministic oracle Turing machine M using A_2 as its oracle, that is, $A_1 = L(M^{A_2})$. (Total recursive functions and oracle Turing machines are defined in Section 5.2). The oracle for A_2 models a hypothetical efficient subroutine for A_2.

If f or M above consumes too much time or space, the reductions they compute are not helpful. To study complexity classes defined by bounds on time and space resources, it is natural to consider resource-bounded reducibilities. Let A_1 and A_2 be languages.

- A_1 is **Karp reducible** to A_2, written $A_1 \le_m^p A_2$, if A_1 is many-one reducible to A_2 via a transformation function that is computable deterministically in polynomial time.
- A_1 is **log-space reducible** to A_2, written $A_1 \le_m^{\log} A_2$, if A_1 is many-one reducible to A_2 via a transformation function that is computable deterministically in $O(\log n)$ space.
- A_1 is **Cook reducible** to A_2, written $A_1 \le_T^p A_2$, if A_1 is Turing reducible to A_2 via a deterministic oracle Turing machine of polynomial time complexity.

The term "polynomial-time reducibility" usually refers to Karp reducibility. If $A_1 \le_m^p A_2$ and $A_2 \le_m^p A_1$, then A_1 and A_2 are **equivalent** under Karp reducibility. Equivalence under Cook reducibility is defined similarly.

Karp and Cook reductions are useful for finding relationships between languages of high complexity, but they are not at all useful for distinguishing between problems in P, because all problems in P are equivalent under Karp (and hence Cook) reductions. (Here and later we ignore the special cases $A_1 = \emptyset$ and $A_1 = \Sigma^*$, and consider them to reduce to any language.)

Log-space reducibility [Jones, 1975] is useful for complexity classes within P, such as NL, for which Karp reducibility allows too many reductions. By definition, for every nontrivial language A_0 (i.e., $A_0 \neq \emptyset$ and $A_0 \neq \Sigma^*$) and for every A in P, necessarily $A \le_m^p A_0$ via a transformation that simply runs a deterministic Turing machine that decides A in polynomial time. It is not known whether log-space reducibility is different from Karp reducibility, however; all transformations for known Karp reductions can be computed in $O(\log n)$ space. Even for decision problems, L is not known to be a proper subset of P.

Theorem 5.6 *Log-space reducibility implies Karp reducibility, which implies Cook reducibility:*

1. *If $A_1 \le_m^{\log} A_2$, then $A_1 \le_m^p A_2$.*
2. *If $A_1 \le_m^p A_2$, then $A_1 \le_T^p A_2$.*

Theorem 5.7 *Log-space reducibility, Karp reducibility, and Cook reducibility are transitive:*

1. *If $A_1 \le_m^{\log} A_2$ and $A_2 \le_m^{\log} A_3$, then $A_1 \le_m^{\log} A_3$.*
2. *If $A_1 \le_m^p A_2$ and $A_2 \le_m^p A_3$, then $A_1 \le_m^p A_3$.*
3. *If $A_1 \le_T^p A_2$ and $A_2 \le_T^p A_3$, then $A_1 \le_T^p A_3$.*

The key property of Cook and Karp reductions is that they preserve polynomial-time feasibility. Suppose $A_1 \le_m^p A_2$ via a transformation f. If M_2 decides A_2, and M_f computes f, then to decide whether an input word x is in A_1, we can use M_f to compute $f(x)$, and then run M_2 on input $f(x)$. If the time complexities of M_2 and M_f are bounded by polynomials t_2 and t_f, respectively, then on each input x of length $n = |x|$, the time taken by this method of deciding A_1 is at most $t_f(n) + t_2(t_f(n))$, which is also a polynomial in n. In summary, if A_2 is feasible, and there is an efficient reduction from A_1 to A_2, then A_1 is feasible. Although this is a simple observation, this fact is important enough to state as a theorem (Theorem 5.8). First, however, we need the concept of "closure."

A class of languages $\mathcal{C}$ is **closed under a reducibility** $\le_r$ if for all languages A_1 and A_2, whenever $A_1 \le_r A_2$ and $A_2 \in \mathcal{C}$, necessarily $A_1 \in \mathcal{C}$.

Theorem 5.8

1. P *is closed under log-space reducibility, Karp reducibility, and Cook reducibility.*
2. NP *is closed under log-space reducibility and Karp reducibility.*
3. L *and* NL *are closed under log-space reducibility.*

We shall see the importance of closure under a reducibility in conjunction with the concept of completeness, which we define in the next section.

5.5.2 Complete Languages

Let $\mathcal{C}$ be a class of languages that represent computational problems. A language A_0 is $\mathcal{C}$-**hard** under a reducibility $\leq_r$ if for all A in $\mathcal{C}$, $A \leq_r A_0$. A language A_0 is $\mathcal{C}$-**complete** under $\leq_r$ if A_0 is $\mathcal{C}$-hard and $A_0 \in \mathcal{C}$. Informally, if A_0 is $\mathcal{C}$-hard, then A_0 represents a problem that is at least as difficult to solve as any problem in $\mathcal{C}$. If A_0 is $\mathcal{C}$-complete, then in a sense, A_0 is one of the most difficult problems in $\mathcal{C}$.

There is another way to view completeness. Completeness provides us with tight lower bounds on the complexity of problems. If a language A is complete for complexity class $\mathcal{C}$, then we have a lower bound on its complexity. Namely, A is as hard as the most difficult problem in $\mathcal{C}$, assuming that the complexity of the reduction itself is small enough not to matter. The lower bound is tight because A is in $\mathcal{C}$; that is, the upper bound matches the lower bound.

In the case $\mathcal{C} = \mathsf{NP}$, the reducibility $\leq_r$ is usually taken to be Karp reducibility unless otherwise stated. Thus, we say

- A language A_0 is **NP-hard** if A_0 is NP-hard under Karp reducibility.
- A_0 is **NP-complete** if A_0 is NP-complete under Karp reducibility.

However, many sources take the term "NP-hard" to refer to Cook reducibility.

Many important languages are now known to be NP-complete. Before we get to them, let us discuss some implications of the statement "A_0 is NP-complete," and also some things this statement does not mean.

The first implication is that *if* there exists a deterministic Turing machine that decides A_0 in polynomial time — that is, if $A_0 \in \mathsf{P}$ — then because P is closed under Karp reducibility (Theorem 5.8 in Section 5.5.1), it would follow that $\mathsf{NP} \subseteq \mathsf{P}$, hence $\mathsf{P} = \mathsf{NP}$. In essence, the question of whether P is the same as NP comes down to the question of whether any particular NP-complete language is in P. Put another way, *all* of the NP-complete languages stand or fall together: if one is in P, then all are in P; if one is not, then all are not. Another implication, which follows by a similar closure argument applied to $\mathsf{co\text{-}NP}$, is that if $A_0 \in \mathsf{co\text{-}NP}$, then $\mathsf{NP} = \mathsf{co\text{-}NP}$. It is also believed unlikely that $\mathsf{NP} = \mathsf{co\text{-}NP}$, as was noted in connection with whether all tautologies have short proofs in Section 5.4.3.

A common misconception is that the above property of NP-complete languages is actually their definition, namely: if $A \in \mathsf{NP}$ and $A \in \mathsf{P}$ implies $\mathsf{P} = \mathsf{NP}$, then A is NP-complete. This "definition" is wrong if $\mathsf{P} \neq \mathsf{NP}$. A theorem due to Ladner [1975] shows that $\mathsf{P} \neq \mathsf{NP}$ if and only if there exists a language A' in $\mathsf{NP} - \mathsf{P}$ such that A' is not NP-complete. Thus, if $\mathsf{P} \neq \mathsf{NP}$, then A' is a counterexample to the "definition."

Another common misconception arises from a misunderstanding of the statement "If A_0 is NP-complete, then A_0 is one of the most difficult problems in NP." This statement is true on one level: if there is any problem at all in NP that is not in P, then the NP-complete language A_0 is one such problem. However, note that there are NP-complete problems in $\mathsf{NTIME}[n]$ — and these problems are, in some sense, much *simpler* than many problems in $\mathsf{NTIME}[n^{10^{500}}]$.

5.5.3 Cook-Levin Theorem

Interest in NP-complete problems started with a theorem of Cook [1971] that was proved independently by Levin [1973]. Recall that SAT is the language of Boolean formulas $\phi(z_1, \ldots, z_r)$ such that there exists a truth assignment to the variables $z_1, \ldots, z_r$ that makes ϕ true.

Theorem 5.9 (Cook-Levin Theorem) SAT *is NP-complete.*

Proof We know already that SAT is in NP, so to prove that SAT is NP-complete, we need to take an arbitrary given language A in NP and show that $A \leq_m^p$ SAT. Take N to be a nondeterministic Turing

machine that decides A in polynomial time. Then the relation $R(x, y) =$ "y is a computation path of N that leads it to accept x" is decidable in deterministic polynomial time depending only on $n = |x|$. We can assume that the length m of possible y's encoded as binary strings depends only on n and not on a particular x.

It is straightforward to show that there is a polynomial p and for each n a Boolean circuit C_n^R with $p(n)$ wires, with $n + m$ input wires labeled $x_1, \ldots, x_n, y_1, \ldots, y_m$ and one output wire w_0, such that $C_n^R(x, y)$ outputs 1 if and only if $R(x, y)$ holds. (We describe circuits in more detail below, and state a theorem for this principle as part 1. of Theorem 5.14.) Importantly, C_n^R itself can be designed in time polynomial in n, and by the universality of NAND, may be composed entirely of binary NAND gates. Label the wires by variables $x_1, \ldots, x_n, y_1, \ldots, y_m, w_0, w_1, \ldots, w_{p(n)-n-m-1}$. These become the variables of our Boolean formulas. For each NAND gate g with input wires u and v, and for each output wire w of g, write down the subformula

$$\phi_{g,w} = (u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$$

This subformula is satisfied by precisely those assignments to u, v, w that give $w = u$ NAND v. The conjunction ϕ_0 of $\phi_{g,w}$ over the polynomially many gates g and their output wires w thus is satisfied only by assignments that set every gate's output correctly given its inputs. Thus, for any binary strings x and y of lengths n, m, respectively, the formula $\phi_1 = \phi_0 \wedge w_0$ is satisfiable by a setting of the wire variables $w_0, w_1, \ldots, w_{p(n)-n-m-1}$ if and only if $C_n^R(x, y) = 1$ — that is, if and only if $R(x, y)$ holds.

Now given any fixed x and taking $n = |x|$, the Karp reduction computes ϕ_1 via C_n^R and ϕ_0 as above, and finally outputs the Boolean formula ϕ obtained by substituting the bit-values of x into ϕ_1. This ϕ has variables $y_1, \ldots, y_m, w_0, w_1, \ldots, w_{p(n)-n-m-1}$, and the computation of ϕ from x runs in deterministic polynomial time. Then $x \in A$ if and only if N accepts x, if and only if there exists y such that $R(x, y)$ holds, if and only if there exists an assignment to the variables $w_0, w_1, \ldots, w_{p(n)-n-m-1}$ *and* $y_1, \ldots, y_m$ that satisfies ϕ, if and only if $\phi \in$ SAT. This shows $A \leq_m^p$ SAT. □

We have actually proved that SAT remains NP-complete even when the given instances ϕ are *restricted* to Boolean formulas that are a conjunction of *clauses*, where each clause consists of (here, at most three) disjuncted literals. Such formulas are said to be in *conjunctive normal form*. Theorem 5.9 is also commonly known as Cook's Theorem.

5.5.4 Proving NP-Completeness

After one language has been proved complete for a class, others can be proved complete by constructing transformations. For NP, if A_0 is NP-complete, then to prove that another language A_1 is NP-complete, it suffices to prove that $A_1 \in$ NP, and to construct a polynomial-time transformation that establishes $A_0 \leq_m^p A_1$. Since A_0 is NP-complete, for every language A in NP, $A \leq_m^p A_0$, hence, by transitivity (Theorem 5.7), $A \leq_m^p A_1$.

Beginning with Cook [1971] and Karp [1972], hundreds of computational problems in many fields of science and engineering have been proved to be NP-complete, almost always by reduction from a problem that was previously known to be NP-complete. The following NP-complete decision problems are frequently used in these reductions — the language corresponding to each problem is the set of instances whose answers are yes.

- 3-SATISFIABILITY (3SAT)
  Instance: A Boolean expression ϕ in conjunctive normal form with three literals per clause [e.g., $(w \vee x \vee \overline{y}) \wedge (\overline{x} \vee y \vee z)$].
  Question: Is ϕ satisfiable?

- VERTEX COVER

  Instance: A graph G and an integer k.

  Question: Does G have a set W of k vertices such that every edge in G is incident on a vertex of W?

- CLIQUE

  Instance: A graph G and an integer k.

  Question: Does G have a set K of k vertices such that every two vertices in K are adjacent in G?

- HAMILTONIAN CIRCUIT

  Instance: A graph G.

  Question: Does G have a circuit that includes every vertex exactly once?

- THREE-DIMENSIONAL MATCHING

  Instance: Sets W, X, Y with $|W| = |X| = |Y| = q$ and a subset $S \subseteq W \times X \times Y$.

  Question: Is there a subset $S' \subseteq S$ of size q such that no two triples in S' agree in any coordinate?

- PARTITION

  Instance: A set S of positive integers.

  Question: Is there a subset $S' \subseteq S$ such that the sum of the elements of S' equals the sum of the elements of $S - S'$?

Note that our ϕ in the above proof of the Cook-Levin Theorem already meets a form of the definition of 3SAT relaxed to allow "at most 3 literals per clause." Padding ϕ with some extra variables to bring up the number in each clause to exactly three, while preserving whether the formula is satisfiable or not, is not difficult, and establishes the NP-completeness of 3SAT. Here is another example of an NP-completeness proof, for the following decision problem:

- TRAVELING SALESMAN PROBLEM (TSP)

  Instance: A set of m "cities" $C_1, \ldots, C_m$, with an integer distance $d(i, j)$ between every pair of cities C_i and C_j, and an integer D.

  Question: Is there a tour of the cities whose total length is at most D, that is, a permutation $c_1, \ldots, c_m$ of $\{1, \ldots, m\}$, such that

$$d(c_1, c_2) + \cdots + d(c_{m-1}, c_m) + d(c_m, c_1) \leq D?$$

First, it is easy to see that TSP is in NP: a nondeterministic Turing machine simply guesses a tour and checks that the total length is at most D.

Next, we construct a reduction from Hamiltonian Circuit to TSP. (The reduction goes from the known NP-complete problem, Hamiltonian Circuit, to the new problem, TSP, not vice versa.)

From a graph G on m vertices $v_1, \ldots, v_m$, define the distance function d as follows:

$$d(i, j) = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge in } G \\ m + 1 & \text{otherwise.} \end{cases}$$

Set $D = m$. Clearly, d and D can be computed in polynomial time from G. Each vertex of G corresponds to a city in the constructed instance of TSP.

If G has a Hamiltonian circuit, then the length of the tour that corresponds to this circuit is exactly m. Conversely, if there is a tour whose length is at most m, then each step of the tour must have distance 1, not $m + 1$. Thus, each step corresponds to an edge of G, and the corresponding sequence of vertices in G is a Hamiltonian circuit.

5.5.5 Complete Problems for Other Classes

Besides NP, the following canonical complexity classes have natural complete problems. The three problems now listed are complete for their respective classes under log-space reducibility.

- NL: GRAPH ACCESSIBILITY PROBLEM

  Instance: A directed graph G with nodes $1, \ldots, N$.

  Question: Does G have a directed path from node 1 to node N?
- P: CIRCUIT VALUE PROBLEM

  Instance: A Boolean circuit (see Section 5.9) with output node u, and an assignment I of $\{\mathbf{0}, \mathbf{1}\}$ to each input node.

  Question: Is $\mathbf{1}$ the value of u under I?
- PSPACE: QUANTIFIED BOOLEAN FORMULAS

  Instance: A Boolean expression with all variables quantified with either $\forall$ or $\exists$ [e.g., $\forall x \forall y \exists z(x \wedge (\overline{y} \vee z))$].

  Question: Is the expression `true`?

These problems can be used to prove other problems are NL-complete, P-complete, and PSPACE-complete, respectively.

Stockmeyer and Meyer [1973] defined a natural decision problem that they proved to be complete for NE. If this problem were in P, then by closure under Karp reducibility (Theorem 5.8), we would have NE $\subseteq$ P, a contradiction of the hierarchy theorems (Theorem 5.4). Therefore, this decision problem is infeasible: it has no polynomial-time algorithm. In contrast, decision problems in NEXP $-$ P that have been constructed by diagonalization are artificial problems that nobody would want to solve anyway. Although diagonalization produces unnatural problems by itself, the combination of diagonalization and completeness shows that *natural* problems are intractable.

The next section points out some limitations of current diagonalization techniques.

5.6 Relativization of the P vs. NP Problem

Let A be a language. Define P^A (respectively, NP^A) to be the class of languages accepted in polynomial time by deterministic (nondeterministic) oracle Turing machines with oracle A.

Proofs that use the diagonalization technique on Turing machines without oracles generally carry over to oracle Turing machines. Thus, for instance, the proof of the DTIME hierarchy theorem also shows that, for *any* oracle A, $\mathsf{DTIME}^A[n^2]$ is properly contained in $\mathsf{DTIME}^A[n^3]$. This can be seen as a *strength* of the diagonalization technique because it allows an argument to "relativize" to computation carried out relative to an oracle. In fact, there are examples of lower bounds (for deterministic, "unrelativized" circuit models) that make crucial use of the fact that the time hierarchies relativize in this sense.

But it can also be seen as a weakness of the diagonalization technique. The following important theorem demonstrates why.

Theorem 5.10 *There exist languages A and B such that $\mathsf{P}^A = \mathsf{NP}^A$, and $\mathsf{P}^B \neq \mathsf{NP}^B$* [Baker et al., 1975].

This shows that resolving the P vs. NP question requires techniques that do not relativize, that is, that do not apply to oracle Turing machines too. Thus, diagonalization as we currently know it is unlikely to succeed in separating P from NP because the diagonalization arguments we know (and in fact *most* of the arguments we know) relativize. Important non-relativizing proof techniques have appeared only recently, in connection with interactive proof systems (Section 5.11.1).

5.7 The Polynomial Hierarchy

Let $\mathcal{C}$ be a class of languages. Define:

- $\mathsf{NP}^{\mathcal{C}} = \bigcup_{A \in \mathcal{C}} \mathsf{NP}^{A}$
- $\Sigma_0^P = \Pi_0^P = \mathsf{P}$

and for $k \geq 0$, define:

- $\Sigma_{k+1}^P = \mathsf{NP}^{\Sigma_k^P}$
- $\Pi_{k+1}^P = \text{co-}\Sigma_{k+1}^P$.

Observe that $\Sigma_1^P = \mathsf{NP}^{\mathsf{P}} = \mathsf{NP}$ because each of polynomially many queries to an oracle language in P can be answered directly by a (nondeterministic) Turing machine in polynomial time. Consequently, $\Pi_1^P = \text{co-}\mathsf{NP}$. For each k, $\Sigma_k^P \cup \Pi_k^P \subseteq \Sigma_{k+1}^P \cap \Pi_{k+1}^P$, but this inclusion is not known to be strict. See Figure 5.3.

The classes Σ_k^P and Π_k^P constitute the **polynomial hierarchy**. Define:

$$\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k^P .$$

It is straightforward to prove that $\mathsf{PH} \subseteq \mathsf{PSPACE}$, but it is not known whether the inclusion is strict. In fact, if $\mathsf{PH} = \mathsf{PSPACE}$, then the polynomial hierarchy collapses to some level, that is, $\mathsf{PH} = \Sigma_m^P$ for some m. In the next section, we define the polynomial hierarchy in two other ways, one of which is in terms of alternating Turing machines.

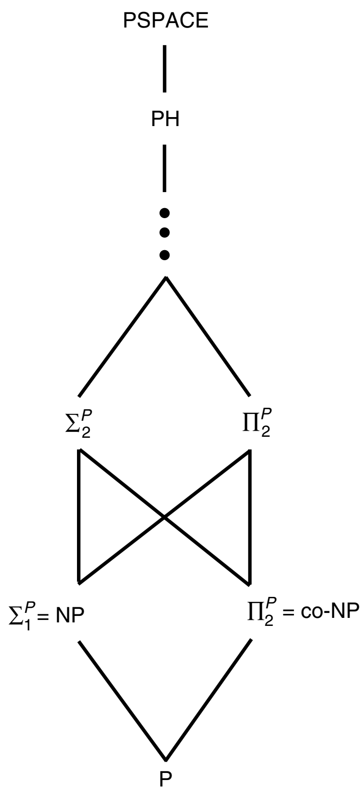


FIGURE 5.3 The polynomial hierarchy.

5.8 Alternating Complexity Classes

In this section, we define time and space complexity classes for alternating Turing machines, and we show how these classes are related to the classes introduced already. The possible computations of an alternating Turing machine M on an input word x can be represented by a tree T_x in which the root is the initial configuration, and the children of a nonterminal node C are the configurations reachable from C by one step of M. For a word x in $L(M)$, define an **accepting subtree** S of T_x to be a subtree of T_x with the following properties:

- S is finite.
- The root of S is the initial configuration with input word x.
- If S has an existential configuration C, then S has exactly one child of C in T_x; if S has a universal configuration C, then S has all children of C in T_x.
- Every leaf is a configuration whose state is the accepting state q_A.

Observe that each node in S is an accepting configuration.

We consider only alternating Turing machines that always halt. For $x \in L(M)$, define the time taken by M to be the height of the shortest accepting tree for x, and the space to be the maximum number of non-blank worktape cells among configurations in the accepting tree that minimizes this number. For $x \notin L(M)$, define the time to be the height of T_x, and the space to be the maximum number of non-blank worktape cells among configurations in T_x.

Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function. Define the following complexity classes:

- $\mathsf{ATIME}[t(n)]$ is the class of languages decided by alternating Turing machines of time complexity $O(t(n))$.
- $\mathsf{ASPACE}[s(n)]$ is the class of languages decided by alternating Turing machines of space complexity $O(s(n))$.

Because a nondeterministic Turing machine is a special case of an alternating Turing machine, for every $t(n)$ and $s(n)$, $\mathsf{NTIME}[t] \subseteq \mathsf{ATIME}[t]$ and $\mathsf{NSPACE}[s] \subseteq \mathsf{ASPACE}[s]$. The next theorem states further relationships between computational resources used by alternating Turing machines, and resources used by deterministic and nondeterministic Turing machines.

Theorem 5.11 (Alternation Theorems) [Chandra et al., 1981]. *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{ATIME}[s(n)^2]$
(b) $\mathsf{ATIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$
(c) $\mathsf{ASPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$
(d) $\mathsf{DTIME}[t(n)] \subseteq \mathsf{ASPACE}[\log t(n)]$

In other words, space on deterministic and nondeterministic Turing machines is polynomially related to time on alternating Turing machines. Space on alternating Turing machines is exponentially related to time on deterministic Turing machines. The following corollary is immediate.

Theorem 5.12

(a) $\mathsf{ASPACE}[O(\log n)] = \mathsf{P}$
(b) $\mathsf{ATIME}[n^{O(1)}] = \mathsf{PSPACE}$
(c) $\mathsf{ASPACE}[n^{O(1)}] = \mathsf{EXP}$

In Section 5.7, we defined the classes of the polynomial hierarchy in terms of oracles, but we can also define them in terms of alternating Turing machines with restrictions on the number of alternations

between existential and universal states. Define a *k-alternating Turing machine* to be a machine such that on every computation path, the number of changes from an existential state to universal state, or from a universal state to an existential state, is at most $k - 1$. Thus, a nondeterministic Turing machine, which stays in existential states, is a 1-alternating Turing machine.

Theorem 5.13 [Stockmeyer, 1976; Wrathall, 1976]. *For any language A, the following are equivalent:*

1. $A \in \Sigma_k^P$.
2. *A is decided in polynomial time by a k-alternating Turing machine that starts in an existential state.*
3. *There exists a language B in* P *and a polynomial p such that for all x, x ∈ A if and only if*

$$(\exists y_1 : |y_1| \le p(|x|))(\forall y_2 : |y_2| \le p(|x|)) \cdots (Qy_k : |y_k| \le p(|x|))[(x, y_1, \ldots, y_k) \in B]$$

where the quantifier Q is ∃ if k is odd, ∀ if k is even.

Alternating Turing machines are closely related to Boolean circuits, which are defined in the next section.

5.9 Circuit Complexity

The hardware of electronic digital computers is based on digital logic gates, connected into combinational circuits (see Chapter 16). Here, we specify a model of computation that formalizes the combinational circuit.

A *Boolean circuit* on n input variables $x_1, \ldots, x_n$ is a directed acyclic graph with exactly n input nodes of indegree 0 labeled $x_1, \ldots, x_n$, and other nodes of indegree 1 or 2, called *gates*, labeled with the Boolean operators in $\{\wedge, \vee, \neg\}$. One node is designated as the output of the circuit. See Figure 5.4. Without loss of generality, we assume that there are no extraneous nodes; there is a directed path from each node to the output node. The indegree of a gate is also called its *fan-in*.

An *input assignment* is a function I that maps each variable x_i to either 0 or 1. The value of each gate g under I is obtained by applying the Boolean operation that labels g to the values of the immediate predecessors of g. The function computed by the circuit is the value of the output node for each input assignment.

A Boolean circuit computes a finite function: a function of only n binary input variables. To decide membership in a language, we need a circuit for each input length n.

A *circuit family* is an infinite set of circuits $C = \{c_1, c_2, \ldots\}$ in which each c_n is a Boolean circuit on n inputs. C *decides* a language $A \subseteq \{\mathbf{0},\mathbf{1}\}^*$ if for every n and every assignment $a_1, \ldots, a_n$ of $\{\mathbf{0},\mathbf{1}\}$ to the n inputs, the value of the output node of c_n is $\mathbf{1}$ if and only if the word $a_1 \cdots a_n \in A$. The *size complexity* of C is the function $z(n)$ that specifies the number of nodes in each c_n. The *depth complexity* of C is the function $d(n)$ that specifies the length of the longest directed path in c_n. Clearly, since the fan-in of each

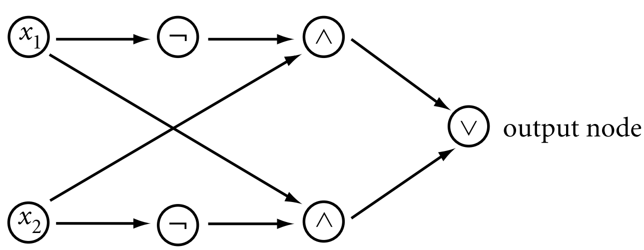


FIGURE 5.4 A Boolean circuit.

gate is at most 2, $d(n) \geq \log z(n) \geq \log n$. The class of languages decided by polynomial-size circuits is denoted by $\mathsf{P/poly}$.

With a different circuit for each input length, a circuit family could solve an undecidable problem such as the halting problem (see Chapter 6). For each input length, a table of all answers for machine descriptions of that length could be encoded into the circuit. Thus, we need to restrict our circuit families. The most natural restriction is that all circuits in a family should have a concise, uniform description, to disallow a different answer table for each input length. Several uniformity conditions have been studied, and the following is the most convenient.

A circuit family $\{c_1, c_2, \ldots\}$ of size complexity $z(n)$ is *log-space uniform* if there exists a deterministic Turing machine M such that on each input of length n, machine M produces a description of c_n, using space $O(\log z(n))$.

Now we define complexity classes for uniform circuit families and relate these classes to previously defined classes. Define the following complexity classes:

- $\mathsf{SIZE}[z(n)]$ is the class of languages decided by log-space uniform circuit families of size complexity $O(z(n))$.
- $\mathsf{DEPTH}[d(n)]$ is the class of languages decided by log-space uniform circuit families of depth complexity $O(d(n))$.

In our notation, $\mathsf{SIZE}[n^{O(1)}]$ equals P, which is a proper subclass of $\mathsf{P/poly}$.

Theorem 5.14

1. *If $t(n)$ is a time-constructible function, then* $\mathsf{DTIME}[t(n)] \subseteq \mathsf{SIZE}[t(n) \log t(n)]$ [Pippenger and Fischer, 1979].
2. $\mathsf{SIZE}[z(n)] \subseteq \mathsf{DTIME}[z(n)^{O(1)}]$.
3. *If $s(n)$ is a space-constructible function and $s(n) \geq \log n$, then* $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DEPTH}[s(n)^2]$ [Borodin, 1977].
4. *If $d(n) \geq \log n$, then* $\mathsf{DEPTH}[d(n)] \subseteq \mathsf{DSPACE}[d(n)]$ [Borodin, 1977].

The next theorem shows that size and depth on Boolean circuits are closely related to space and time on alternating Turing machines, provided that we permit sublinear running times for alternating Turing machines, as follows. We augment alternating Turing machines with a random-access input capability. To access the cell at position j on the input tape, M writes the binary representation of j on a special tape, in $\log j$ steps, and enters a special reading state to obtain the symbol in cell j.

Theorem 5.15 [Ruzzo, 1979]. *Let $t(n) \geq \log n$ and $s(n) \geq \log n$ be such that the mapping $n \mapsto (t(n), s(n))$ (in binary) is computable in time $s(n)$.*

1. *Every language decided by an alternating Turing machine of simultaneous space complexity $s(n)$ and time complexity $t(n)$ can be decided by a log-space uniform circuit family of simultaneous size complexity $2^{O(s(n))}$ and depth complexity $O(t(n))$.*
2. *If $d(n) \geq (\log z(n))^2$, then every language decided by a log-space uniform circuit family of simultaneous size complexity $z(n)$ and depth complexity $d(n)$ can be decided by an alternating Turing machine of simultaneous space complexity $O(\log z(n))$ and time complexity $O(d(n))$.*

In a sense, the Boolean circuit family is a model of parallel computation, because all gates compute independently, in parallel. For each $k \geq 0$, NC^k denotes the class of languages decided by log-space uniform bounded fan-in circuits of polynomial size and depth $O((\log n)^k)$, and AC^k is defined analogously for unbounded fan-in circuits. In particular, AC^k is the same as the class of languages decided by a parallel machine model called the CRCW PRAM with polynomially many processors in parallel time $O((\log n)^k)$ [Stockmeyer and Vishkin, 1984].

5.10 Probabilistic Complexity Classes

Since the 1970s, with the development of randomized algorithms for computational problems (see Chapter 12). Complexity theorists have placed randomized algorithms on a firm intellectual foundation. In this section, we outline some basic concepts in this area.

A **probabilistic Turing machine** M can be formalized as a nondeterministic Turing machine with exactly two choices at each step. During a computation, M chooses each possible next step with independent probability 1/2. Intuitively, at each step, M flips a fair coin to decide what to do next. The probability of a computation path of t steps is $1/2^t$. The probability that M accepts an input string x, denoted by $p_M(x)$, is the sum of the probabilities of the accepting computation paths.

Throughout this section, we consider only machines whose time complexity $t(n)$ is time-constructible. Without loss of generality, we can assume that every computation path of such a machine halts in exactly t steps.

Let A be a language. A probabilistic Turing machine M decides A with

		for all $x \in A$	for all $x \notin A$
unbounded two-sided error	if	$p_M(x) > 1/2$	$p_M(x) \leq 1/2$
bounded two-sided error	if	$p_M(x) > 1/2 + \epsilon$	$p_M(x) < 1/2 - \epsilon$
		for some positive constant ϵ	
one-sided error	if	$p_M(x) > 1/2$	$p_M(x) = 0$

Many practical and important probabilistic algorithms make one-sided errors. For example, in the primality testing algorithm of Solovay and Strassen [1977], when the input x is a prime number, the algorithm *always* says "prime"; when x is composite, the algorithm *usually* says "composite," but may occasionally say "prime." Using the definitions above, this means that the Solovay-Strassen algorithm is a one-sided error algorithm for the set A of composite numbers. It also is a bounded two-sided error algorithm for $\overline{A}$, the set of prime numbers.

These three kinds of errors suggest three complexity classes:

1. PP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with unbounded two-sided error.
2. BPP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with bounded two-sided error.
3. RP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with one-sided error.

In the literature, RP is also called R.

A probabilistic Turing machine M is a PP-**machine** (respectively, a BPP-**machine**, an RP-**machine**) if M has polynomial time complexity, and M decides with two-sided error (bounded two-sided error, one-sided error).

Through repeated Bernoulli trials, we can make the error probabilities of BPP-machines and RP-machines arbitrarily small, as stated in the following theorem. (Among other things, this theorem implies that RP $\subseteq$ BPP.)

Theorem 5.16 *If $A \in$ BPP, then for every polynomial $q(n)$, there exists a BPP-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every $x \in A$, and $p_M(x) < 1/2^{q(n)}$ for every $x \notin A$.*

If $L \in$ RP, then for every polynomial $q(n)$, there exists an RP-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every x in L.

It is important to note just how minuscule the probability of error is (provided that the coin flips are truly random). If the probability of error is less than $1/2^{5000}$, then it is less likely that the algorithm produces an incorrect answer than that the computer will be struck by a meteor. An algorithm whose probability of

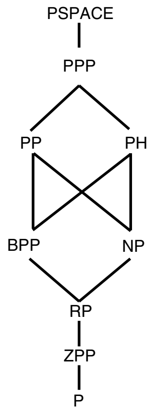


FIGURE 5.5 Probabilistic complexity classes.

error is $1/2^{5000}$ is essentially as good as an algorithm that makes no errors. For this reason, many computer scientists consider BPP to be the class of practically feasible computational problems.

Next, we define a class of problems that have probabilistic algorithms that make no errors. Define:

$$\mathsf{ZPP} = \mathsf{RP} \cap \text{co-}\mathsf{RP}$$

The letter Z in ZPP is for zero probability of error, as we now demonstrate. Suppose $A \in \mathsf{ZPP}$. Here is an algorithm that checks membership in A. Let M be an RP-machine that decides A, and let M' be an RP-machine that decides $\overline{A}$. For an input string x, alternately run M and M' on x, repeatedly, until a computation path of one machine accepts x. If M accepts x, then accept x; if M' accepts x, then reject x. This algorithm works correctly because when an RP-machine accepts its input, it does not make a mistake. This algorithm might not terminate, but with very high probability, the algorithm terminates after a few iterations.

The next theorem expresses some known relationships between probabilistic complexity classes and other complexity classes, such as classes in the polynomial hierarchy. See Section 5.7 and Figure 5.5.

Theorem 5.17

(a) $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{PP} \subseteq \mathsf{PSPACE}$ [Gill, 1977]
(b) $\mathsf{RP} \subseteq \mathsf{NP} \subseteq \mathsf{PP}$ [Gill, 1977]
(c) $\mathsf{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$ [Lautemann, 1983; Sipser, 1983]
(d) $\mathsf{BPP} \subset \mathsf{P}/\mathsf{poly}$
(e) $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$ [Toda, 1991]

An important recent research area called **de-randomization** studies whether randomized algorithms can be converted to deterministic ones of the same or comparable efficiency. For example, if there is a language in E that requires Boolean circuits of size $2^{\Omega(n)}$ to decide it, then $\mathsf{BPP} = \mathsf{P}$ [Impagliazzo and Wigderson, 1997].

5.11 Interactive Models and Complexity Classes

5.11.1 Interactive Proofs

In Section 5.3.2, we characterized NP as the set of languages whose membership proofs can be checked quickly, by a deterministic Turing machine M of polynomial time complexity. A different notion of proof involves interaction between two parties, a prover P and a verifier V, who exchange messages. In an **interactive proof system** [Goldwasser et al., 1989], the prover is an all-powerful machine, with

unlimited computational resources, analogous to a teacher. The verifier is a computationally limited machine, analogous to a student. Interactive proof systems are also called "Arthur-Merlin games": the wizard Merlin corresponds to P, and the impatient Arthur corresponds to V [Babai and Moran, 1988].

Formally, an **interactive proof system** comprises the following:

- A read-only input tape on which an input string x is written.
- A *verifier* V, which is a probabilistic Turing machine augmented with the capability to send and receive messages. The running time of V is bounded by a polynomial in $|x|$.
- A *prover* P, which receives messages from V and sends messages to V.
- A tape on which V writes messages to send to P, and a tape on which P writes messages to send to V. The length of every message is bounded by a polynomial in $|x|$.

A computation of an interactive proof system (P, V) proceeds in rounds, as follows. For $j = 1, 2, \ldots,$ in round j, V performs some steps, writes a message m_j, and temporarily stops. Then P reads m_j and responds with a message m'_j, which V reads in round $j + 1$. An interactive proof system (P, V) **accepts** an input string x if the probability of acceptance by V satisfies $p_V(x) > 1/2$.

In an interactive proof system, a prover can convince the verifier about the truth of a statement without exhibiting an entire proof, as the following example illustrates.

Consider the graph non-isomorphism problem: the input consists of two graphs G and H, and the decision is `yes` if and only if G is not isomorphic to H. Although there is a short proof that two graphs *are* isomorphic (namely: the proof consists of the isomorphism mapping G onto H), nobody has found a general way of proving that two graphs are *not* isomorphic that is significantly shorter than listing all $n!$ permutations and showing that each fails to be an isomorphism. (That is, the graph non-isomorphism problem is in co-NP, but is not known to be in NP.) In contrast, the verifier V in an interactive proof system is able to take statistical evidence into account, and determine "beyond all reasonable doubt" that two graphs are non-isomorphic, using the following protocol.

In each round, V randomly chooses either G or H with equal probability; if V chooses G, then V computes a random permutation G' of G, presents G' to P, and asks P whether G' came from G or from H (and similarly if V chooses H). If P gave an erroneous answer on the first round, and G is isomorphic to H, then after k subsequent rounds, the probability that P answers all the subsequent queries correctly is $1/2^k$. (To see this, it is important to understand that the prover P does not see the coins that V flips in making its random choices; P sees only the graphs G' and H' that V sends as messages.) V accepts the interaction with P as "proof" that G and H are non-isomorphic if P is able to pick the correct graph for 100 consecutive rounds. Note that V has ample grounds to accept this as a convincing demonstration: if the graphs are indeed isomorphic, the prover P would have to have an incredible streak of luck to fool V.

It is important to comment that de-randomization techniques applied to these proof systems have shown that under plausible hardness assumptions, proofs of non-isomorphism of sub-exponential length (or even polynomial length) do exist [Klivans and van Melkebeek, 2002]. Thus, many complexity theoreticians now conjecture that the graph isomorphism problem lies in NP $\cap$ co-NP.

The complexity class IP comprises the languages A for which there exists a verifier V and a positive ϵ such that

- There exists a prover $\hat{P}$ such that for all x in A, the interactive proof system $(\hat{P}, V)$ accepts x with probability greater than $1/2 + \epsilon$; and
- For every prover P and every $x \notin A$, the interactive proof system (P, V) rejects x with probability greater than $1/2 + \epsilon$.

By substituting random choices for existential choices in the proof that $\mathsf{ATIME}(t) \subseteq \mathsf{DSPACE}(t)$ (Theorem 5.11), it is straightforward to show that $\mathsf{IP} \subseteq \mathsf{PSPACE}$. It was originally believed likely that IP was a small subclass of PSPACE. Evidence supporting this belief was the construction of an oracle language B for which $\text{co-NP}^B - \mathsf{IP}^B \neq \emptyset$ [Fortnow and Sipser, 1988], so that IP^B is strictly included in PSPACE^B. Using a proof technique that does not relativize, however, Shamir [1992] proved that, in fact, IP and PSPACE are the same class.

Theorem 5.18 $\mathsf{IP} = \mathsf{PSPACE}$. [Shamir, 1992].

If NP is a proper subset of PSPACE, as is widely believed, then Theorem 5.18 says that interactive proof systems can decide a larger class of languages than NP.

5.11.2 Probabilistically Checkable Proofs

In an interactive proof system, the verifier does not need a complete conventional proof to become convinced about the membership of a word in a language, but uses random choices to query parts of a proof that the prover may know. This interpretation inspired another notion of "proof": a proof consists of a (potentially) large amount of information that the verifier need only inspect in a few places in order to become convinced. The following definition makes this idea more precise.

A language A has a **probabilistically checkable proof** if there exists an oracle BPP-machine M such that:

- For all $x \in A$, there exists an oracle language B_x such that M^{B_x} accepts x with probability 1.
- For all $x \notin A$, and for every language B, machine M^B accepts x with probability strictly less than 1/2.

Intuitively, the oracle language B_x represents a proof of membership of x in A. Notice that B_x can be finite since the length of each possible query during a computation of M^{B_x} on x is bounded by the running time of M. The oracle language takes the role of the prover in an interactive proof system — but in contrast to an interactive proof system, the prover cannot change strategy adaptively in response to the questions that the verifier poses. This change results in a potentially stronger system, since a machine M that has bounded error probability relative to all languages B might not have bounded error probability relative to some adaptive prover. Although this change to the proof system framework may seem modest, it leads to a characterization of a class that seems to be much larger than PSPACE.

Theorem 5.19 *A has a probabilistically checkable proof if and only if $A \in$* NEXP [Babai et al., 1991].

Although the notion of probabilistically checkable proofs seems to lead us away from feasible complexity classes, by considering natural restrictions on how the proof is accessed, we can obtain important insights into familiar complexity classes.

Let $\mathsf{PCP}[r(n), q(n)]$ denote the class of languages with probabilistically checkable proofs in which the probabilistic oracle Turing machine M makes $O[r(n)]$ random binary choices, and queries its oracle $O[q(n)]$ times. (For this definition, we assume that M has either one or two choices for each step.) It follows from the definitions that $\mathsf{BPP} = \mathsf{PCP}(n^{O(1)}, 0)$, and $\mathsf{NP} = \mathsf{PCP}(0, n^{O(1)})$.

Theorem 5.20 (The PCP Theorem) $\mathsf{NP} = \mathsf{PCP}[\emptyset \log n, \emptyset(1)]$ [Arora et al., 1998].

Theorem 5.20 asserts that for every language A in NP, a proof that $x \in A$ can be encoded so that the verifier can be convinced of the correctness of the proof (or detect an incorrect proof) by using only $O(\log n)$ random choices, and inspecting only a *constant* number of bits of the proof.

5.12 Kolmogorov Complexity

Until now, we have considered only dynamic complexity measures, namely, the time and space used by Turing machines. Kolmogorov complexity is a static complexity measure that captures the difficulty of describing a string. For example, the string consisting of three million zeroes can be described with fewer than three million symbols (as in this sentence). In contrast, for a string consisting of three million randomly generated bits, with high probability there is no shorter description than the string itself.

Let U be a universal Turing machine (see Section 5.2.3). Let λ denote the empty string. The **Kolmogorov complexity** of a binary string y with respect to U, denoted by $K_U(y)$, is the length of the shortest binary string i such that on input $\langle i, \lambda \rangle$, machine U outputs y. In essence, i is a description of y, for it tells U how to generate y.

The next theorem states that different choices for the universal Turing machine affect the definition of Kolmogorov complexity in only a small way.

Theorem 5.21 (Invariance Theorem) *There exists a universal Turing machine U such that for every universal Turing machine U', there is a constant c such that for all y, $K_U(y) \leq K_{U'}(y) + c$.*

Henceforth, let K be defined by the universal Turing machine of Theorem 5.21. For every integer n and every binary string y of length n, because y can be described by giving itself explicitly, $K(y) \leq n + c'$ for a constant c'. Call y **incompressible** if $K(y) \geq n$. Since there are 2^n binary strings of length n and only $2^n - 1$ possible shorter descriptions, there exists an incompressible string for every length n.

Kolmogorov complexity gives a precise mathematical meaning to the intuitive notion of "randomness." If someone flips a coin 50 times and it comes up "heads" each time, then intuitively, the sequence of flips is not random — although from the standpoint of probability theory, the all-heads sequence is precisely as likely as any other sequence. Probability theory does not provide the tools for calling one sequence "more random" than another; Kolmogorov complexity theory does.

Kolmogorov complexity provides a useful framework for presenting combinatorial arguments. For example, when one wants to prove that an object with some property P exists, then it is sufficient to show that any object that does *not* have property P has a short description; thus, any incompressible (or "random") object must have property P. This sort of argument has been useful in proving lower bounds in complexity theory.

5.13 Research Issues and Summary

The core research questions in complexity theory are expressed in terms of separating complexity classes:

- Is L different from NL?
- Is P different from RP or BPP?
- Is P different from NP?
- Is NP different from PSPACE?

Motivated by these questions, much current research is devoted to efforts to understand the power of nondeterminism, randomization, and interaction. In these studies, researchers have gone well beyond the theory presented in this chapter:

- Beyond Turing machines and Boolean circuits, to restricted and specialized models in which nontrivial lower bounds on complexity can be proved
- Beyond deterministic reducibilities, to nondeterministic and probabilistic reducibilities, and refined versions of the reducibilities considered here
- Beyond worst-case complexity, to average-case complexity

Recent research in complexity theory has had direct applications to other areas of computer science and mathematics. Probabilistically checkable proofs were used to show that obtaining approximate solutions to some optimization problems is as difficult as solving them exactly. Complexity theory has provided new tools for studying questions in finite model theory, a branch of mathematical logic. Fundamental questions in complexity theory are intimately linked to practical questions about the use of cryptography for computer security, such as the existence of one-way functions and the strength of public key cryptosystems.

This last point illustrates the urgent practical need for progress in computational complexity theory. Many popular cryptographic systems in current use are based on unproven assumptions about the difficulty

of computing certain functions (such as the factoring and discrete logarithm problems). All of these systems are thus based on wishful thinking and conjecture. Research is needed to resolve these open questions and replace conjecture with mathematical certainty.

Acknowledgments

Donna Brown, Bevan Das, Raymond Greenlaw, Lane Hemaspaandra, John Jozwiak, Sung-il Pae, Leonard Pitt, Michael Roman, and Martin Tompa read earlier versions of this chapter and suggested numerous helpful improvements. Karen Walny checked the references.

Eric W. Allender was supported by the National Science Foundation under Grant CCR-0104823. Michael C. Loui was supported by the National Science Foundation under Grant SES-0138309. Kenneth W. Regan was supported by the National Science Foundation under Grant CCR-9821040.

Defining Terms

Complexity class: A set of languages that are decided within a particular resource bound. For example, $\mathsf{NTIME}(n^2 \log n)$ is the set of languages decided by nondeterministic Turing machines within $O(n^2 \log n)$ time.

Constructibility: A function $f(n)$ is time (respectively, space) constructible if there exists a deterministic Turing machine that halts after exactly $f(n)$ steps (after using exactly $f(n)$ worktape cells) for every input of length n.

Diagonalization: A technique for constructing a language A that differs from every $L(M_i)$ for a list of machines $M_1, M_2, \ldots$.

NP-complete: A language A_0 is NP-complete if $A_0 \in \mathsf{NP}$ and $A \leq_m^p A_0$ for every A in NP; that is, for every A in NP, there exists a function f computable in polynomial time such that for every x, $x \in A$ if and only if $f(x) \in A_0$.

Oracle: An oracle is a language A to which a machine presents queries of the form "Is w in A" and receives each correct answer in one step.

Padding: A technique for establishing relationships between complexity classes that uses padded versions of languages, in which each word is padded out with multiple occurrences of a new symbol — the word x is replaced by the word $x\#^{f(|x|)}$ for a numeric function f — in order to artificially reduce the complexity of the language.

Reduction: A language A reduces to a language B if a machine that decides B can be used to decide A efficiently.

Time and space complexity: The time (respectively, space) complexity of a deterministic Turing machine M is the maximum number of steps taken (nonblank cells used) by M among all input words of length n.

Turing machine: A Turing machine M is a model of computation with a read-only input tape and multiple worktapes. At each step, M reads the tape cells on which its access heads are located, and depending on its current state and the symbols in those cells, M changes state, writes new symbols on the worktape cells, and moves each access head one cell left or right or not at all.

References

Allender, E., Loui, M.C., and Regan, K.W. 1999. Chapter 27: Complexity classes, Chapter 28: Reducibility and completeness, Chapter 29: Other complexity classes and measures. In *Algorithms and Theory of Computation Handbook*, Ed. M. J. Atallah, CRC Press, Boca Raton, FL.

Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M. 1998. Proof verification and hardness of approximation problems. *J. ACM*, 45(3):501–555.

Babai, L. and Moran, S. 1988. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. *J. Comput. Sys. Sci.*, 36(2):254–276.

Babai, L., Fortnow, L., and Lund, C. 1991. Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40.

Baker, T., Gill, J., and Solovay, R. 1975. Relativizations of the P = NP? question. *SIAM J. Comput.*, 4(4): 431–442.

Balcázar, J.L., Díaz, J., and Gabarró, J. 1990. *Structural Complexity II*. Springer-Verlag, Berlin.

Balcázar, J.L., Díaz, J., and Gabarró, J. 1995. *Structural Complexity I*. 2nd ed. Springer-Verlag, Berlin.

Book, R.V. 1974. Comparing complexity classes. *J. Comp. Sys. Sci.*, 9(2):213–229.

Borodin, A. 1972. Computational complexity and the existence of complexity gaps. *J. Assn. Comp. Mach.*, 19(1):158–174.

Borodin, A. 1977. On relating time and space to size and depth. *SIAM J. Comput.*, 6(4):733–744.

Bovet, D.P. and Crescenzi, P. 1994. *Introduction to the Theory of Complexity*. Prentice Hall International Ltd; Hertfordshire, U.K.

Chandra, A.K., Kozen, D.C., and Stockmeyer, L.J. 1981. Alternation. *J. Assn. Comp. Mach.*, 28(1):114–133.

Cook, S.A. 1971. The complexity of theorem-proving procedures. In *Proc. 3rd Annu. ACM Symp. Theory Comput.*, pp. 151–158. Shaker Heights, OH.

Du, D-Z. and Ko, K.-I. 2000. *Theory of Computational Complexity*. Wiley, New York.

Fortnow, L. and Sipser, M. 1988. Are there interactive protocols for co-NP languages? *Inform. Process. Lett.*, 28(5):249–251.

Garey, M.R. and Johnson, D.S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco.

Gill, J. 1977. Computational complexity of probabilistic Turing machines. *SIAM J. Comput.*, 6(4):675–695.

Goldwasser, S., Micali, S., and Rackoff, C. 1989. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208.

Hartmanis, J., Ed. 1989. *Computational Complexity Theory.* American Mathematical Society, Providence, RI.

Hartmanis, J. 1994. On computational complexity and the nature of computer science. *Commun. ACM*, 37(10):37–43.

Hartmanis, J. and Stearns, R.E. 1965. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306.

Hemaspaandra, L.A. and Ogihara, M. 2002. *The Complexity Theory Companion*. Springer, Berlin.

Hemaspaandra, L.A. and Selman, A.L., Eds. 1997. *Complexity Theory Retrospective II*. Springer, New York.

Hennie, F. and Stearns, R.A. 1966. Two–way simulation of multitape Turing machines. *J. Assn. Comp. Mach.*, 13(4):533–546.

Immerman, N. 1988. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938.

Impagliazzo, R. and Wigderson, A. 1997. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. *Proc. 29th Annu. ACM Symp. Theory Comput.*, ACM Press, pp. 220–229. El Paso, TX.

Jones, N.D. 1975. Space-bounded reducibility among combinatorial problems. *J. Comp. Sys. Sci.*, 11(1):68–85. Corrigendum *J. Comp. Sys. Sci.*, 15(2):241, 1977.

Karp, R.M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations.* R.E. Miller and J.W. Thatcher, Eds., pp. 85–103. Plenum Press, New York.

Klivans, A.R. and van Melkebeek, D. 2002. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM J. Comput.*, 31(5):1501–1526.

Ladner, R.E. 1975. On the structure of polynomial-time reducibility. *J. Assn. Comp. Mach.*, 22(1):155–171.

Lautemann, C. 1983. BPP and the polynomial hierarchy. *Inf. Proc. Lett.*, 17(4):215–217.

Levin, L. 1973. Universal search problems. *Problems of Information Transmission*, 9(3):265–266 (in Russian).

Li, M. and Vitányi, P.M.B. 1997. *An Introduction to Kolmogorov Complexity and Its Applications*. 2nd ed. Springer-Verlag, New York.

Papadimitriou, C.H. 1994. *Computational Complexity*. Addison-Wesley, Reading, MA.

Pippenger, N. and Fischer, M. 1979. Relations among complexity measures. *J. Assn. Comp. Mach.*, 26(2):361–381.
Ruzzo, W.L. 1981. On uniform circuit complexity. *J. Comp. Sys. Sci.*, 22(3):365–383.
Savitch, W.J. 1970. Relationship between nondeterministic and deterministic tape complexities. *J. Comp. Sys. Sci.*, 4(2):177–192.
Seiferas, J.I., Fischer, M.J., and Meyer, A.R. 1978. Separating nondeterministic time complexity classes. *J. Assn. Comp. Mach.*, 25(1):146–167.
Shamir, A. 1992. IP = PSPACE. *J. ACM* 39(4):869–877.
Sipser, M. 1983. Borel sets and circuit complexity. In *Proc. 15th Annual ACM Symposium on the Theory of Computing*, pp. 61–69.
Sipser, M. 1992. The history and status of the P versus NP question. In *Proc. 24th Annu. ACM Symp. Theory Comput.*, ACM Press, pp. 603–618. Victoria, B.C., Canada.
Solovay, R. and Strassen, V. 1977. A fast Monte-Carlo test for primality. *SIAM J. Comput.*, 6(1):84–85.
Stearns, R.E. 1990. Juris Hartmanis: the beginnings of computational complexity. In *Complexity Theory Retrospective.* A.L. Selman, Ed., pp. 5–18, Springer-Verlag, New York.
Stockmeyer, L.J. 1976. The polynomial time hierarchy. *Theor. Comp. Sci.*, 3(1):1–22.
Stockmeyer, L.J. 1987. Classifying the computational complexity of problems. *J. Symb. Logic,* 52:1–43.
Stockmeyer, L.J. and Chandra, A.K. 1979. Intrinsically difficult problems. *Sci. Am.*, 240(5):140–159.
Stockmeyer, L.J. and Meyer, A.R. 1973. Word problems requiring exponential time: preliminary report. In *Proc. 5th Annu. ACM Symp. Theory Comput.*, ACM Press, pp. 1–9. Austin, TX.
Stockmeyer, L.J. and Vishkin, U. 1984. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422.
Szelepcsényi, R. 1988. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284.
Toda, S. 1991. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877.
van Leeuwen, J. 1990. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier Science, Amsterdam, and M.I.T. Press, Cambridge, MA.
Wagner, K. and Wechsung, G. 1986. *Computational Complexity.* D. Reidel, Dordrecht, The Netherlands.
Wrathall, C. 1976. Complete sets and the polynomial-time hierarchy. *Theor. Comp. Sci.*, 3(1):23–33.

Further Information

This chapter is a short version of three chapters written by the same authors for the *Algorithms and Theory of Computation Handbook* [Allender et al., 1999].

The formal theoretical study of computational complexity began with the paper of Hartmanis and Stearns [1965], who introduced the basic concepts and proved the first results. For historical perspectives on complexity theory, see Hartmanis [1994], Sipser [1992], and Stearns [1990].

Contemporary textbooks on complexity theory are by Balcázar et al. [1990, 1995], Bovet and Crescenzi [1994], Du and Ko [2000], Hemaspaandra and Ogihara [2002], and Papadimitriou [1994]. Wagner and Wechsung [1986] is an exhaustive survey of complexity theory that covers work published before 1986. Another perspective of some of the issues covered in this chapter can be found in the survey by Stockmeyer [1987].

A good general reference is the *Handbook of Theoretical Computer Science* [van Leeuwen, 1990], Volume A. The following chapters in that *Handbook* are particularly relevant: "Machine Models and Simulations," by P. van Emde Boas, pp. 1–66; "A Catalog of Complexity Classes," by D.S. Johnson, pp. 67–161; "Machine-Independent Complexity Theory," by J.I. Seiferas, pp. 163–186; "Kolmogorov Complexity and its Applications," by M. Li and P.M.B. Vitányi, pp. 187–254; and "The Complexity of Finite Functions," by R.B. Boppana and M. Sipser, pp. 757–804, which covers circuit complexity.

A collection of articles edited by Hartmanis [1989] includes an overview of complexity theory, and chapters on sparse complete languages, on relativizations, on interactive proof systems, and on applications of complexity theory to cryptography. A collection edited by Hemaspaandra and Selman [1997] includes chapters on quantum and biological computing, on proof systems, and on average case complexity.

For specific topics in complexity theory, the following references are helpful. Garey and Johnson [1979] explain NP-completeness thoroughly, with examples of NP-completeness proofs, and a collection of hundreds of NP-complete problems. Li and Vitányi [1997] provide a comprehensive, scholarly treatment of Kolmogorov complexity, with many applications.

Surveys and lecture notes on complexity theory that can be obtained via the Web are maintained by A. Czumaj and M. Kutylowski at:

`http://www.uni-paderborn.de/fachbereich/AG/agmadh/WWW/english/scripts.html`

As usual with the Web, such links are subject to change. Two good stem pages to begin searches are the site for SIGACT (the ACM Special Interest Group on Algorithms and Computation Theory) and the site for the annual IEEE Conference on Computational Complexity:

`http://sigact.acm.org/`
`http://www.computationalcomplexity.org/`

The former site has a pointer to a "Virtual Address Book" that indexes the personal Web pages of over 1000 computer scientists, including all three authors of this chapter. Many of these pages have downloadable papers and links to further research resources. The latter site includes a pointer to the *Electronic Colloquium on Computational Complexity* maintained at the University of Trier, Germany, which includes downloadable prominent research papers in the field, often with updates and revisions.

Research papers on complexity theory are presented at several annual conferences, including the annual ACM Symposium on Theory of Computing; the annual International Colloquium on Automata, Languages, and Programming, sponsored by the European Association for Theoretical Computer Science (EATCS); and the annual Symposium on Foundations of Computer Science, sponsored by the IEEE. The annual Conference on Computational Complexity (formerly Structure in Complexity Theory), also sponsored by the IEEE, is entirely devoted to complexity theory. Research articles on complexity theory regularly appear in the following journals, among others: *Chicago Journal on Theoretical Computer Science, Computational Complexity, Information and Computation, Journal of the ACM, Journal of Computer and System Sciences, SIAM Journal on Computing, Theoretical Computer Science,* and *Theory of Computing Systems* (formerly *Mathematical Systems Theory*). Each issue of *ACM SIGACT News* and *Bulletin of the EATCS* contains a column on complexity theory.

6

Formal Models and Computability

Tao Jiang
University of California

Ming Li
University of Waterloo

Bala Ravikumar
University of Rhode Island

6.1 Introduction

The concept of **algorithms** is perhaps almost as old as human civilization. The famous Euclid's algorithm is more than 2000 years old. Angle trisection, solving diophantine equations, and finding polynomial roots in terms of radicals of coefficients are some well-known examples of algorithmic questions. However, until the 1930s the notion of algorithms was used informally (or rigorously but in a limited context). It was a major triumph of logicians and mathematicians of this century to offer a rigorous definition of this fundamental concept. The revolution that resulted in this triumph was a collective achievement of many mathematicians, notably Church, Gödel, Kleene, Post, and Turing. Of particular interest is a machine model proposed by Turing in 1936, which has come to be known as a **Turing machine** [Turing 1936].

This particular achievement had numerous significant consequences. It led to the concept of a general-purpose computer or universal computation, a revolutionary idea originally anticipated by Babbage in the 1800s. It is widely acknowledged that the development of a universal Turing machine was prophetic of the modern all-purpose digital computer and played a key role in the thinking of pioneers in the development of modern computers such as von Neumann [Davis 1980]. From a mathematical point of view, however, a more interesting consequence was that it was now possible to show the *nonexistence* of algorithms, hitherto impossible due to their elusive nature. In addition, many apparently different definitions of an algorithm proposed by different researchers in different continents turned out to be equivalent (in a precise technical sense, explained later). This equivalence led to the widely held hypothesis known as the *Church–Turing thesis* that mechanical solvability is the same as solvability on a Turing machine.

Formal languages are closely related to algorithms. They were introduced as a way to convey mathematical proofs without errors. Although the concept of a formal language dates back at least to the time of Leibniz, a systematic study of them did not begin until the beginning of this century. It became a vigorous field of study when Chomsky formulated simple grammatical rules to describe the syntax of a language

[Chomsky 1956]. **Grammars** and **formal languages** entered into computability theory when Chomsky and others found ways to use them to classify algorithms.

The main theme of this chapter is about formal models, which include Turing machines (and their variants) as well as grammars. In fact, the two concepts are intimately related. Formal computational models are aimed at providing a framework for computational problem solving, much as electromagnetic theory provides a framework for problems in electrical engineering. Thus, formal models guide the way to build computers and the way to program them. At the same time, new models are motivated by advances in the technology of computing machines. In this chapter, we will discuss only the most basic computational models and use these models to classify problems into some fundamental classes. In doing so, we hope to provide the reader with a conceptual basis with which to read other chapters in this Handbook.

6.2 Computability and a Universal Algorithm

Turing's notion of mechanical computation was based on identifying the basic steps of such computations. He reasoned that an operation such as multiplication is not primitive because it can be divided into more basic steps such as digit-by-digit multiplication, shifting, and adding. Addition itself can be expressed in terms of more basic steps such as add the lowest digits, compute, carry, and move to the next digit, etc. Turing thus reasoned that the most basic features of mechanical computation are the abilities to read and write on a storage medium (which he chose to be a linear tape divided into cells or squares) and to make some simple logical decisions. He also restricted each tape cell to hold only one among a finite number of symbols (which we call the *tape alphabet*).* The decision step enables the computer to control the sequence of actions. To make things simple, Turing restricted the next action to be performed on a cell neighboring the one on which the current action occurred. He also introduced an instruction that told the computer to stop. In summary, Turing proposed a model to characterize mechanical computation as being carried out as a sequence of instructions of the form: write a symbol (such as 0 or 1) on the tape cell, move to the next cell, observe the symbol currently scanned and choose the next step accordingly, or stop.

These operations define a language we call the GOTO language.** Its instructions are

PRINT i (i is a tape symbol)
GO RIGHT
GO LEFT
GO TO STEP j IF i IS SCANNED
STOP

A **program** in this language is a sequence of instructions (written one per line) numbered $1 - k$. To run a program written in this language, we should provide the *input*. We will assume that the input is a string of symbols from a finite input alphabet (which is a subset of the tape alphabet), which is stored on the tape before the computation begins. How much memory should we allow the computer to use? Although we do not want to place any bounds on it, allowing an infinite tape is not realistic. This problem is circumvented by allowing *expandable memory*. In the beginning, the tape containing the input defines its boundary. When the machine moves beyond the current boundary, a new memory cell will be attached with a special symbol B (blank) written on it. Finally, we define the result of computation as the contents of the tape when the computer reaches the STOP instruction.

We will present an example program written in the GOTO language. This program accomplishes the simple task of doubling the number of 1s (Figure 6.1). More precisely, on the input containing k 1s, the

*This bold step of using a discrete model was perhaps the harbinger of the digital revolution that was soon to follow.

**Turing's original formulation is closer to our presentation in Section 6.5. But the GOTO language presents an equivalent model.

```
1   PRINT 0
2   GO LEFT
3   GO TO STEP 2 IF 1 IS SCANNED
4   PRINT 1
5   GO RIGHT
6   GO TO STEP 5 IF 1 IS SCANNED
7   PRINT 1
8   GO RIGHT
9   GO TO STEP 1 IF 1 IS SCANNED
10  STOP
```

FIGURE 6.1 The doubling program in the GOTO language.

program produces $2k$ 1s. Informally, the program achieves its goal as follows. When it reads a 1, it changes the 1 to 0, moves left looking for a new cell, writes a 1 in the cell, returns to the starting cell and rewrites as 1, and repeats this step for each 1. Note the way the GOTO instructions are used for repetition. This feature is the most important aspect of programming and can be found in all of the imperative style programming languages.

The simplicity of the GOTO language is rather deceptive. There is strong reason to believe that it is powerful enough that any mechanical computation can be expressed by a suitable program in the GOTO language. Note also that the programs written in the GOTO language may not always halt, that is, on certain inputs, the program may never reach the STOP instruction. In this case, we say that the output is undefined.

We can now give a precise definition of what an algorithm is. An algorithm is any program written in the GOTO language with the additional property that it halts on all inputs. Such programs will be called *halting programs*. Throughout this chapter, we will be interested mainly in computational problems of a special kind called *decision problems* that have a yes/no answer. We will modify our language slightly when dealing with decision problems. We will augment our instruction set to include ACCEPT and REJECT (and omit STOP). When the ACCEPT (REJECT) instruction is reached, the machine will output yes or 1 (no or 0) and halt.

6.2.1 Some Computational Problems

We will temporarily shift our focus from the tool for problem solving (the computer) to the problems themselves. Throughout this chapter, a computational problem refers to an input/output relationship. For example, consider the problem of squaring an integer input. This problem assigns to each integer (such as 22) its square (in this case 484). In technical terms, this input/output relationship defines a function. Therefore, solving a computational problem is the same as computing the function defined by the problem. When we say that an algorithm (or a program) solves a problem, what we mean is that, for all inputs, the program halts and produces the correct output. We will allow inputs of arbitrary size and place no restrictions. A reader with primary interest in software applications is apt to question the validity (or even the meaningfulness) of allowing inputs of arbitrary size because it makes the set of all *possible* inputs infinite, and thus unrealistic, in real-world programming. But there are no really good alternatives. Any finite bound is artificial and is likely to become obsolete as the technology and our requirements change. Also, in practice, we do not know how to take advantage of restrictions on the size of the inputs. (See the discussion about nonuniform models in Section 6.5.) Problems (functions) that can be solved by an algorithm (or a halting GOTO program) are called *computable*.

As already remarked, we are interested mainly in decision problems. A decision problem is said to be decidable if there is a halting GOTO program that solves it correctly on all inputs. An important class of problems called **partially decidable decision problems** can be defined by relaxing our requirement a little bit; a decision problem is partially decidable if there is a GOTO program that halts and outputs 1 on all inputs for which the output should be 1 and either halts and outputs 0 or loops forever on the other inputs.

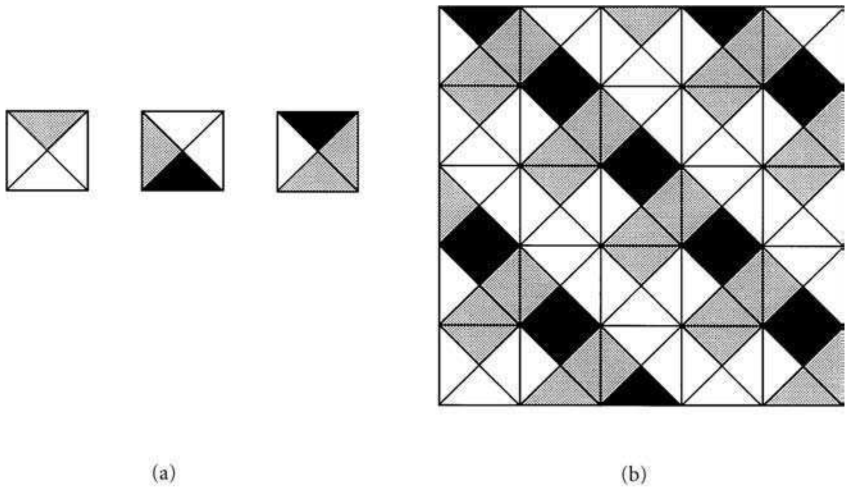


FIGURE 6.2 An example of tiling.

This means that the program may never give a wrong answer but is not required to halt on negative inputs (i.e., inputs with 0 as output).

We now list some problems that are fundamental either because of their inherent importance or because of their historical roles in the development of computation theory:

Problem 1 (halting problem). The input to this problem is a program P in the GOTO language and a binary string x. The expected output is 1 (or yes) if the program P halts when run on the input x, 0 (or no) otherwise.

Problem 2 (universal computation problem). A related problem takes as input a program P and an input x and produces as output what (if any) P would produce on input x. (Note that this is a decision problem if P is restricted to a yes/no program.)

Problem 3 (string compression). For a string x, we want to find the shortest program in the GOTO language that when started with the empty tape (i.e., tape containing one B symbol) halts and prints x. Here shortest means the total number of symbols in the program is as small as possible.

Problem 4 (tiling). A tile* is a square card of unit size (i.e., 1×1) divided into four quarters by two diagonals, each quarter colored with some color (selected from a finite set of colors). The tiles have fixed orientation and cannot be rotated. Given some finite set T of such tiles as input, the program is to determine if finite rectangular areas of all sizes (i.e., $k \times m$ for all positive integers k and m) can be tiled using only the given tiles such that the colors on any two touching edges are the same. It is assumed that an unlimited number of cards of each type is available. Figure 6.2(b) shows how the base set of tiles given in Figure 6.2(a) can be used to tile a 5×5 square area.

Problem 5 (linear programming). Given a system of linear inequalities (called constraints), such as $3x - 4y \leq 13$ with integer coefficients, the goal is to find if the system has a solution satisfying all of the constraints.

Some remarks must be made about the preceding problems. The problems in our list include nonnumerical problems and *meta problems*, which are problems about other problems. The first two problems are motivated by a quest for reliable program design. An algorithm for problem 1 (if it exists) can be used to test if a program contains an infinite loop. Problem 2 is motivated by an attempt to design a **universal**

*More precisely, a Wang tile, after Hao Wang, who wrote the first research paper on it.

algorithm, which can simulate any other. This problem was first attempted by Babbage, whose analytical engine had many ingredients of a modern electronic computer (although it was based on mechanical devices). Problem 3 is an important problem in information theory and arises in the following setting. Physical theories are aimed at creating simple laws to explain large volumes of experimental data. A famous example is Kepler's laws, which explained Tycho Brahe's huge and meticulous observational data. Problem 3 asks if this compression process can be automated. When we allow the inference rules to be sufficiently strong, this problem becomes **undecidable**. We will not discuss this problem further in this section but will refer the reader to some related formal systems discussed in Li and Vitányi [1993]. The tiling problem is not merely an interesting puzzle. It is an art form of great interest to architects and painters. Tiling has recently found applications in crystallography. Linear programming is a problem of central importance in economics, game theory, and operations research.

In the remainder of the section, we will present some basic algorithm design techniques and sketch how these techniques can be used to solve some of the problems listed (or their special cases). The main purpose of this discussion is to present techniques for showing the decidability (or partial decidability) of these problems. The reader can learn more advanced techniques of algorithm design in some later sections of this chapter as well as in many later chapters of this volume.

6.2.1.1 Table Lookup

The basic idea is to create a table for a function f, which needs to be computed by tabulating in one column an input x and the corresponding $f(x)$ in a second column. Then the table itself can be used as an algorithm. This method cannot be used directly because the set of all inputs is infinite. Therefore, it is not very useful, although it can be made to work in conjunction with the technique described subsequently.

6.2.1.2 Bounding the Search Domain

The difficulty of establishing the decidability of a problem is usually caused by the fact that the object we are searching for may have no known upper limit. Thus, if we can place such an upper bound (based on the structure of the problem), then we can reduce the search to a finite domain. Then table lookup can be used to complete the search (although there may be better methods in practice). For example, consider the following special case of the tiling problem: Let k be a fixed integer, say 1000. Given a set of tiles, we want to determine whether all rectangular rooms of shape $k \times n$ can be tiled for all n. (Note the difference between this special case and the general problem. The general one allows k and n both to have unbounded value. But here we allow only n to be unbounded.) It can be shown (see Section 6.5 for details) that there are two bounds n_0 and n_1 (they depend on k) such that if there is at least one tile of size $k \times t$ that can be tiled for some $n_0 \leq t \leq n_1$ then every tile of size $k \times n$ can be tiled. If no $k \times t$ tile can be tiled for any t between n_0 and n_1, then obviously the answer is no. Thus, we have reduced an infinite search domain to a finite one.

As another example, consider the linear programming problem. The set of possible solutions to this problem is infinite, and thus a table search cannot be used. But it is possible to reduce the search domain to a finite set using the geometric properties of the set of solutions of the linear programming problem. The fact that the set of solutions is convex makes the search especially easy.

6.2.1.3 Use of Subroutines

This is more of a program design tool than a tool for algorithm design. A central concept of programming is repetitive (or iterative) computation. We already observed how GOTO statements can be used to perform a sequence of steps repetitively. The idea of a subroutine is another central concept of programming. The idea is to make use of a program P itself as a single step in another program Q. Building programs from simpler programs is a natural way to deal with the complexity of programming tasks. We will illustrate the idea with a simple example. Consider the problem of multiplying two positive integers i and j. The input to the problem will be the form $11 \ldots 1011 \ldots 1$ (i 1s followed by a 0, followed by j 1s) and the output will be $i * j$ 1s (with possibly some 0s on either end). We will use the notation 1^i01^j to denote the starting configuration of the tape. This just means that the tape contains i 1s followed by a 0 followed by j 1s.

TABLE 6.1 Coding the GOTO Instructions

Instruction	Code
PRINT i	0001^{i+1}
GO LEFT	001
GO RIGHT	010
GO TO j IF i IS SCANNED	0111^j01^{i+1}
STOP	100

The basic idea behind a GOTO program for this problem is simple; add j 1s on the right end of tape exactly $i - 1$ times and then erase the original sequence of i 1s on the left. A little thought reveals that the subroutine we need here is to duplicate a string of 1s so that if we start with $x02^k1^j$ a call to the subroutine will produce $x02^{k+j}1^j$. Here x is just any sequence of symbols. Note the role played by the symbol 2. As new 1s are created on the right, the old 1s change to 2s. This will ensure that there are exactly j 1s on the right end of the tape all of the time. This duplication subroutine is very similar to the doubling program, and the reader should have very little difficulty writing this program. Finally, the multiplication program can be done using the copy subroutine $(i - 1)$ times.

6.2.2 A Universal Algorithm

We will now present in some detail a (partial) solution to problem 2 by arguing that there is a program U written in the GOTO language, which takes as input a program P (also written using the GOTO language) and an input x and produces as output $P(x)$, the output of P on input x. For convenience, we will assume that all programs written in the GOTO language use a fixed alphabet containing just 0, 1, and B. Because we have assumed this for all programs in the GOTO language, we should first address the issue of how an input to program U will look. We cannot directly place a program P on the tape because the alphabet used to write the program P uses letters G, O, T, O, etc. This minor problem can be easily circumvented by coding. The idea is to represent each instruction using only 0 and 1. One such coding scheme is shown in Table 6.1.

To encode an entire program, we simply write down in order (without the line numbers) the code for each instruction as given in the table. For example, here is the code for the doubling program shown in Figure 6.1:

$$0001001011110110001101001111111011000110100111011100$$

Note that the encoded string contains all of the information about the program so that the encoding is completely reversible. From now on, if P is a program in the GOTO language, then code(P) will denote its binary code as just described. When there is no confusion, we will identify P and code(P). Before proceeding further, the reader may want to test his/her understanding of the encoding/decoding process by decoding the following string: 010011101100.

The basic idea behind the construction of a universal algorithm is simple, although the details involved in actually constructing one are enormous. We will present the central ideas and leave out the actual construction. Such a construction was carried out in complete detail by Turing himself and was simplified by others.* U has as its input code(P) followed by the string x. U simulates the computational steps of P on input x. It divides the input tape into three segments, one containing the program P, the second one essentially containing the contents of the tape of P as it changes with successive moves, and the third one containing the line number in program P of the instruction being currently simulated (similar to a *program counter* in an actual computer).

*A particularly simple exposition can be found in Robinson [1991].

We now describe a *cycle* of computation by U, which is similar to a central processing unit (CPU) cycle in a real computer. A single instruction of P is implemented by U in one cycle. First, U should know which location on the tape that P is currently reading. A simple artifact can handle this as follows: U uses in its tape alphabet two special symbols $0'$ and $1'$. U stores the tape of P in the tape segment alluded to in the previous paragraph exactly as it would appear when the program P is run on the input x with one minor modification. The symbol currently being read by program P is stored as the *primed version* ($0'$ is the primed version of 0, etc.). As an example, suppose after completing 12 instructions, P is reading the fourth symbol (from left) on its tape containing 01001001. Then the tape region of U after 12 cycles looks like $0100'1001$. At the beginning of a new cycle, U uses a subroutine to move to the region of the tape that contains the ith instruction of program P where i is the value of the program counter. It then decodes the ith instruction. Based on what type it is, U proceeds as follows: If it is a PRINT i instruction, then U scans the tape until the unique primed symbol in the *tape region* is reached and rewrites it as instructed. If it is a GO LEFT or GO RIGHT symbol, U locates the primed symbol, unprimes it, and primes its left or right neighbor, as instructed. In both cases, U returns to the program counter and increments it. If the instruction is GO TO i IF j IS SCANNED, U reads the primed symbol, and if it is j', U changes the program counter to i. This completes a cycle. Note that the three regions may grow and contract while U executes the cycles of computation just described. This may result in one of them running into another. U must then shift one of them to the left or right and make room as needed.

It is not too difficult to see that all of the steps described can be done using the instructions of the GOTO language. The main point to remember is that these actions will have to be coded as a single program, which has nothing whatsoever to do with program P. In fact, the program U is totally independent of P. If we replace P with some other program Q, it should simulate Q as well. The preceding argument shows that problem 2 is partially decidable. But it does not show that this problem is decidable. Why? It is because U may not halt on all inputs; specifically, consider an input consisting of a program P and a string x such that P does not halt on x. Then U will also keep executing cycle after cycle the moves of P and will never halt. In fact, in Section 6.3, we will show that problem 2 is not decidable.

6.3 Undecidability

Recall the definition of an undecidable problem. In this section, we will establish the undecidability of Problem 2, Section 6.2. The simplest way to establish the existence of undecidable problems is as follows: There are more problems than there are programs, the former set being uncountable, whereas the latter is countably infinite.* But this argument is purely existential and does not identify any specific problem as undecidable. In what follows, we will show that Problem 2 introduced in Section 6.2 is one such problem.

6.3.1 Diagonalization and Self-Reference

Undecidability is inextricably tied to the concept of self-reference, and so we begin by looking at this rather perplexing and sometimes paradoxical concept. The idea of self-reference seems to be many centuries old and may have originated with a barber in ancient Greece who had a sign board that read: "I shave all those who do not shave themselves." When the statement is applied to the barber himself, we get a self-contradictory statement. Does he shave himself? If the answer is yes, then he is one of those who shaves himself, and so the barber should not shave him. The contrary answer no is equally untenable. So neither yes nor no seems to be the correct answer to the question; this is the essence of the paradox. The barber's

*The reader who does not know what countable and uncountable infinities are can safely ignore this statement; the rest of the section does not depend on it.

paradox has made entry into modern mathematics in various forms. We will present some of them in the next few paragraphs.*

The first version, called Berry's paradox, concerns English descriptions of natural numbers. For example, the number 7 can be described by many different phrases: seven, six plus one, the fourth smallest prime, etc. We are interested in the *shortest* of such descriptions, namely, the one with the fewest letters in it. Clearly there are (infinitely) many positive integers whose shortest descriptions exceed 100 letters. (A simple counting argument can be used to show this. The set of positive integers is infinite, but the set of positive integers with English descriptions in fewer than or equal to 100 letters is finite.) Let D denote the set of positive integers that do not have English descriptions with fewer than 100 letters. Thus, D is not empty. It is a well-known fact in set theory that any nonempty subset of positive integers has a smallest integer. Let x be the smallest integer in D. Does x have an English description with fewer than or equal to 100 letters? By the definition of the set D and x, we have: x is "the smallest positive integer that cannot be described in English in fewer than 100 letters." This is clearly absurd because part of the last sentence in quotes is a description of x and it contains fewer than 100 letters in it. A similar paradox was found by the British mathematician Bertrand Russell when he considered the set of all sets that do not include themselves as elements, that is, $S = \{x \mid x \notin x\}$. The question "Is $S \in S$?" leads to a similar paradox.

As a last example, we will consider a charming self-referential paradox due to mathematician William Zwicker. Consider the collection of all two-person games (such as chess, tic-tac-toe, etc.) in which players make alternate moves until one of them loses. Call such a game *normal* if it has to end in a finite number of moves, no matter what strategies the two players use. For example, tic-tac-toe must end in at most nine moves and so it is normal. Chess is also normal because the 50-move rule ensures that the game cannot go forever. Now here is *hypergame*. In the first move of the hypergame, the first player calls out a normal game, and then the two players go on to play the game, with the second player making the first move. The question is: "Is hypergame normal?" Suppose it is normal. Imagine two players playing hypergame. The first player can call out hypergame (since it is a normal game). This makes the second player call out the name of a normal game, hypergame can be called out again and they can keep saying hypergame without end, and this contradicts the definition of a normal game. On the other hand, suppose it is not a normal game. But now in the first move, player 1 cannot call out hypergame and would call a normal game instead, and so the infinite move sequence just given is not possible, and so hypergame is normal after all!

In the rest of the section, we will show how these paradoxes can be modified to give nonparadoxical but surprising conclusions about the decidability of certain problems. Recall the encoding we presented in Section 6.2 that encodes any program written in the GOTO language as a binary string. Clearly this encoding is reversible in the sense that if we start with a program and encode it, it is possible to decode it back to the program. However, not every binary string corresponds to a program because there are many strings that cannot be decoded in a meaningful way, for example, 11010011000110. For the purposes of this section, however, it would be convenient if we can treat *every* binary string as a program. Thus, we will simply stipulate that any undecodable string be decoded to the program containing the single statement

1. REJECT

In the following discussion, we will identify a string x with a GOTO program to which it decodes. Now define a function f_D as follows: $f_D(x) = 1$ if x, decoded into a GOTO program, does not halt when started with x itself as the input. Note the self-reference in this definition. Although the definition of f_D seems artificial, its importance will become clear in the next section when we use it to show the undecidability of Problem 2. First we will prove that f_D is not computable. Actually, we will prove a stronger statement, namely, that f_D is not even partially decidable. [Recall that a function is partially decidable if there is a GOTO

*The most enchanting discussions of self-reference are due to the great puzzlist and mathematician R. Smullyan who brings out the breadth and depth of this concept in such delightful books as *What is the name of this book?* published by Prentice–Hall in 1978 and *Satan, Cantor, and Infinity* published by Alfred A. Knopf in 1992. We heartily recommend them to anyone who wants to be amused, entertained, and, more importantly, educated on the intricacies of mathematical logic and computability.

program (not necessarily halting) that computes it. An important distinction between computable and semicomputable functions is that a GOTO program for the latter need not halt on inputs with output = 0.]

Theorem 6.1 *Function f_D is not partially decidable.*

The proof is by contradiction. Suppose a GOTO program P' computes the function f_D. We will modify P' into another program P in the GOTO language such that P computes the same function as P' but has the additional property that it will never terminate its computation by ending up in a REJECT statement.* Thus, P is a program with the property that it computes f_D and halts on an input y if and only if $f_D(y) = 1$. We will complete the proof by showing that there is at least one input in which the program produces a wrong output, that is, there is an x such that $f_D(x) \neq P(x)$.

Let x be the encoding of program P. Now consider the question: Does P halt when given x as input? Suppose the answer is yes. Then, by the way we constructed P, here $P(x) = 1$. On the other hand, the definition of f_D implies that $f_D(x) = 0$. (This is the punch line in this proof. We urge the reader to take a few moments and read the definition of f_D a few times and make sure that he or she is convinced about this fact!) Similarly, if we start with the assumption that $P(x) = 0$, we are led to the conclusion that $f_D(x) = 1$. *In both cases,* $f_D(x) \neq P(x)$ and thus P is not the correct program for f_D. Therefore, P' is not the correct program for f_D either because P and P' compute the same function. This contradicts the hypothesis that such a program exists, and the proof is complete.

Note the crucial difference between the paradoxes we presented earlier and the proof of this theorem. Here we do not have a paradox because our conclusion is of the form $f_D(x) = 0$ if and only if $P(x) = 1$ and not $f_D(x) = 1$ if and only if $f_D(x) = 0$. But in some sense, the function f_D was motivated by Russell's paradox. We can similarly create another function f_Z (based on Zwicker's paradox of hypergame). Let f be any function that maps binary strings to $\{0, 1\}$. We will describe a method to generate successive functions f_1, f_2, etc., as follows: Suppose $f(x) = 0$ for all x. Then we cannot create any more functions, and the sequence stops with f. On the other hand, if $f(x) = 1$ for some x, then choose one such x and decode it as a GOTO program. This defines another function; call it f_1 and repeat the same process with f_1 in the place of f. We call f a normal function if no matter how x is selected at each step, the process terminates after a finite number of steps. A simple example of a nonnormal function is as follows: Suppose $P(Q) = 1$ for some program P and input Q and at the same time $Q(P) = 1$ (note that we are using a program and its code interchangeably), then it is easy to see that the functions defined by both P and Q are not normal. Finally, define $f_Z(X) = 1$ if X is a normal program, 0 if it is not. We leave it as an instructive exercise to the reader to show that f_Z is not semicomputable. A perceptive reader will note the connection between Berry's paradox and problem 3 in our list (string compression problem) just as f_Z is related to Zwicker's paradox. Such a reader should be able to show the undecidability of problem 3 by imitating Berry's paradox.

6.3.2 Reductions and More Undecidable Problems

Theory of computation deals not only with the behavior of individual problems but also with relations among them. A **reduction** is a simple way to relate two problems so that we can deduce the (un)decidability of one from the (un)decidability of the other. Reduction is similar to using a subroutine. Consider two problems A and B. We say that problem A can be reduced to problem B if there is an algorithm for B provided that A has one. To define the reduction (also called a *Turing reduction*) precisely, it is convenient to augment the instruction set of the GOTO programming language to include a new instruction CALL X, i, j where X is a (different) GOTO program, and i and j are line numbers. In detail, the execution of such augmented programs is carried out as follows: When the computer reaches the instruction CALL X,

*The modification needed to produce P from P' is straightforward. If P' did not have any REJECT statements at all, then no modification would be needed. If it had, then we would have to replace each one by a looping statement, which keeps repeating the same instruction forever.

i, j, the program will simply start executing the instructions of the program from line 1, treating whatever is on the tape currently as the input to the program X. When (if at all) X finishes the computation by reaching the ACCEPT statement, the execution of the original program continues at line number i and, if it finishes with REJECT, the original program continues from line number j.

We can now give a more precise definition of a reduction between two problems. Let A and B be two computational problems. We say that A is reducible to B if there is a halting program Y in the GOTO language for problem A in which calls can be made to a halting program X for problem B. The algorithm for problem A described in the preceding reduction does not assume the availability of program X and cannot use the details behind the design of this algorithm. The right way to think about a reduction is as follows: Algorithm Y, from time to time, needs to know the solutions to different instances of problem B. It can query an algorithm for problem B (as a black box) and use the answer to the query for making further decisions. An important point to be noted is that the program Y actually can be implemented even if program X was never built as long as someone can correctly answer some questions asked by program Y about the output of problem B for certain inputs. Programs with such calls are sometimes called *oracle programs*. Reduction is rather difficult to assimilate at the first attempt, and so we will try to explain it using a puzzle. How do you play two chess games, one each with Kasparov and Anand (perhaps currently the world's two best players) and ensure that you get at least one point? (You earn one point for a win, 0 for a loss, and 1/2 for a draw.) Because you are a novice and are pitted against two Goliaths, you are allowed a concession. You can choose to play white or black on either board. The well-known answer is the following: Take white against one player, say, Anand, and black against the other, namely, Kasparov. Watch the first move of Kasparov (as he plays white) and make the same move against Anand, get his reply and play it back to Kasparov and keep playing back and forth like this. It takes only a moment's thought that you are guaranteed to win (exactly) 1 point. The point is that your game involves taking the position of one game, applying the algorithm of one player, getting the result and applying it to the other board, etc., and you do not even have to know the rules of chess to do this. This is exactly how algorithm Y is required to use algorithm X.

We will use reductions to show the undecidability as follows: Suppose A can be reduced to B as in the preceding definition. If there is an algorithm for problem B, it can be used to design a program for A by essentially imitating the execution of the augmented program for A (with calls to the oracle for B) as just described. But we will turn it into a negative argument as follows: If A is undecidable, then so is B. Thus, a reduction from a problem known to be undecidable to problem B will prove B's undecidability.

First we define a new problem, Problem 2′, which is a special case of Problem 2. Recall that in Problem 2 the input is (the code of) a program P in GOTO language and a string x. The output required is $P(x)$. In Problem 2′, the input is (only) the code of a program P and the output required is $P(P)$, that is, instead of requiring P to run on a given input, this problem requires that it be run on its own code. This is clearly a special case of problem 2. The reader may readily see the self-reference in Problem 2′ and suspect that it may be undecidable; therefore, the more general Problem 2 may be undecidable as well. We will establish these claims more rigorously as follows.

We first observe a general statement about the decidability of a function f (or problem) and its *complement*. The complement function is defined to take value 1 on all inputs for which the original function value is 0 and vice versa. The statement is that a function f is decidable if and only if the complement $\bar{f}$ is decidable. This can be easily proved as follows. Consider a program P that computes f. Change P into $\bar{P}$ by interchanging all of the ACCEPT and REJECT statements. It is easy to see that $\bar{P}$ actually computes $\bar{f}$. The converse also is easily seen to hold. It readily follows that the function defined by problem 2′ is undecidable because it is, in fact, the complement of f_D.

Finally, we will show that problem 2 is uncomputable. The idea is to use a reduction from problem 2′ to problem 2. (Note the direction of reduction. This always confuses a beginner.) Suppose there is an algorithm for problem 2. Let X be the GOTO language program that implements this algorithm. X takes as input code(P) (for any program P) followed by x, produces the result $P(x)$, and halts. We want to design a program Y that takes as input code(P) and produce the output $P(P)$ using calls to program X. It is clear what needs to be done. We just create the input in proper form code(P) followed by code(P) and call X. This requires first duplicating the input, but this is a simple programming task similar to the

one we demonstrated in our first program in Section 6.2. Then a call to X completes the task. This shows that Problem $2'$ reduces to Problem 2, and thus the latter is undecidable as well.

By a more elaborate reduction (from f_D), it can be shown that tiling is not partially decidable. We will not do it here and refer the interested reader to Harel [1992]. But we would like to point out how the undecidability result can be used to infer a result about tiling. This deduction is of interest because the result is an important one and is hard to derive directly. We need the following definition before we can state the result. A different way to pose the tiling problem is whether a given set of tiles can tile *an entire plane* in such a way that all of the adjacent tiles have the same color on the meeting quarter. (Note that this question is different from the way we originally posed it: Can a given set of tiles tile any *finite* rectangular region? Interestingly, the two problems are identical in the sense that the answer to one version is yes if and only if it is yes for the other version.) Call a tiling of the plane periodic if one can identify a $k \times k$ square such that the entire tiling is made by repeating this $k \times k$ square tile. Otherwise, call it *aperiodic*. Consider the question: Is there a (finite) set of unit tiles that can tile the plane, but only aperiodically? The answer is yes and it can be shown from the total undecidability of the tiling problem. Suppose the answer is no. Then, for any given set of tiles, the entire plane can be tiled if and only if the plane can be tiled periodically. But a periodic tiling can be found, if one exists, by trying to tile a $k \times k$ region for successively increasing values of k. This process will eventually succeed (in a finite number of steps) if the tiling exists. This will make the tiling problem partially decidable, which contradicts the total undecidability of the problem. This means that the assumption that the entire plane can be tiled if and only if some $k \times k$ region can be tiled is wrong. Thus, there exists a (finite) set of tiles that can tile the entire plane, but only aperiodically.

6.4 Formal Languages and Grammars

The universe of strings is probably the most general medium for the representation of information. This section is concerned with sets of strings called *languages* and certain systems generating these languages such as *grammars*. Every programming language including Pascal, C, or Fortran can be precisely described by a grammar. Moreover, the grammar allows us to write a computer program (called the lexical analyzer in a compiler) to determine if a piece of code is syntactically correct in the programming language. Would not it be nice to also have such a grammar for English and a corresponding computer program which can tell us what English sentences are grammatically correct?* The focus of this brief exposition is the formalism and mathematical properties of various languages and grammars. Many of the concepts have applications in domains including natural language and computer language processing, string matching, etc. We begin with some standard definitions about languages.

Definition 6.1 An *alphabet* is a finite nonempty set of *symbols*, which are assumed to be *indivisible*.

For example, the alphabet for English consists of 26 uppercase letters $A, B, \ldots, Z$ and 26 lowercase letters $a, b, \ldots, z$. We usually use the symbol Σ to denote an alphabet.

Definition 6.2 A *string* over an alphabet Σ is a finite sequence of symbols of Σ.

The number of symbols in a string x is called its *length*, denoted $|x|$. It is convenient to introduce an empty string, denoted ϵ, which contains no symbols at all. The length of ϵ is 0.

Definition 6.3 Let $x = a_1 a_2 \cdots a_n$ and $y = b_1 b_2 \cdots b_m$ be two strings. The *concatenation* of x and y, denoted xy, is the string $a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$.

*Actually, English and the other natural languages have grammars; but these grammars are not precise enough to tell apart the correct and incorrect sentences with 100% accuracy. The main problem is that *there is no universal agreement* on what are grammatically correct English sentences.

Thus, for any string x, $\epsilon x = x\epsilon = x$. For any string x and integer $n \geq 0$, we use x^n to denote the string formed by sequentially concatenating n copies of x.

Definition 6.4 The set of all strings over an alphabet Σ is denoted Σ^* and the set of all nonempty strings over Σ is denoted Σ^+. The empty set of strings is denoted $\emptyset$.

Definition 6.5 For any alphabet Σ, a *language* over Σ is a set of strings over Σ. The members of a language are also called the *words* of the language.

Example 6.1

The sets $L_1 = \{01, 11, 0110\}$ and $L_2 = \{0^n 1^n \mid n \geq 0\}$ are two languages over the binary alphabet $\{0, 1\}$. The string 01 is in both languages, whereas 11 is in L_1 but not in L_2.

Because languages are just sets, standard set operations such as union, intersection, and complementation apply to languages. It is useful to introduce two more operations for languages: *concatenation* and *Kleene closure*.

Definition 6.6 Let L_1 and L_2 be two languages over Σ. The concatenation of L_1 and L_2, denoted $L_1 L_2$, is the language $\{xy \mid x \in L_1, y \in L_2\}$.

Definition 6.7 Let L be a language over Σ. Define $L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$. The Kleene closure of L, denoted L^*, is the language

$$L^* = \bigcup_{i \geq 0} L^i$$

and the *positive closure* of L, denoted L^+, is the language

$$L^+ = \bigcup_{i \geq 1} L^i$$

In other words, the Kleene closure of language L consists of all strings that can be formed by concatenating some words from L. For example, if $L = \{0, 01\}$, then $LL = \{00, 001, 010, 0101\}$ and L^* includes all binary strings in which every 1 is preceded by a 0. L^+ is the same as L^* except it excludes ϵ in this case. Note that, for any language L, L^* always contains ϵ and L^+ contains ϵ if and only if L does. Also note that Σ^* is in fact the Kleene closure of the alphabet Σ when viewed as a language of words of length 1, and Σ^+ is just the positive closure of Σ.

6.4.1 Representation of Languages

In general, a language over an alphabet Σ is a subset of Σ^*. How can we describe a language rigorously so that we know if a given string belongs to the language or not? As shown in the preceding paragraphs, a finite language such as L_1 in Example 6.1 can be explicitly defined by enumerating its elements, and a simple infinite language such as L_2 in the same example can be described using a rule characterizing all members of L_2. It is possible to define some more systematic methods to represent a wide class of languages. In the following, we will introduce three such methods: regular expressions, pattern systems, and grammars. The languages that can be described by this kind of system are often referred to as *formal languages*.

Definition 6.8 Let Σ be an alphabet. The *regular expressions* over Σ and the languages they represent are defined inductively as follows.

1. The symbol $\emptyset$ is a regular expression, denoting the empty set.
2. The symbol ϵ is a regular expression, denoting the set $\{\epsilon\}$.

3. For each $a \in \Sigma$, a is a regular expression, denoting the set $\{a\}$.
4. If r and s are regular expressions denoting the languages R and S, then $(r + s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

For example, $((0(0 + 1)^*) + ((0 + 1)^*0))$ is a regular expression over $\{0, 1\}$, and it represents the language consisting of all binary strings that begin or end with a 0. Because the set operations union and concatenation are both associative, many parentheses can be omitted from regular expressions if we assume that Kleene closure has higher precedence than concatenation and concatenation has higher precedence than union. For example, the preceding regular expression can be abbreviated as $0(0 + 1)^* + (0 + 1)^*0$. We will also abbreviate the expression rr^* as r^+. Let us look at a few more examples of regular expressions and the languages they represent.

Example 6.2

The expression $0(0 + 1)^*1$ represents the set of all strings that begin with a 0 and end with a 1.

Example 6.3

The expression $0 + 1 + 0(0 + 1)^*0 + 1(0 + 1)^*1$ represents the set of all nonempty binary strings that begin and end with the same bit.

Example 6.4

The expressions 0^*, 0^*10^*, and $0^*10^*10^*$ represent the languages consisting of strings that contain no 1, exactly one 1, and exactly two 1s, respectively.

Example 6.5

The expressions $(0 + 1)^*1(0 + 1)^*1(0 + 1)^*$, $(0 + 1)^*10^*1(0 + 1)^*$, $0^*10^*1(0 + 1)^*$, and $(0 + 1)^*10^*10^*$ all represent the same set of strings that contain at least two 1s.

For any regular expression r, the language represented by r is denoted as $L(r)$. Two regular expressions representing the same language are called *equivalent*. It is possible to introduce some identities to algebraically manipulate regular expressions to construct equivalent expressions, by tailoring the set identities for the operations union, concatenation, and Kleene closure to regular expressions. For more details, see Salomaa [1966]. For example, it is easy to prove that the expressions $r(s + t)$ and $rs + rt$ are equivalent and $(r^*)^*$ is equivalent to r^*.

Example 6.6

Let us construct a regular expression for the set of all strings that contain no consecutive 0s. A string in this set may begin and end with a sequence of 1s. Because there are no consecutive 0s, every 0 that is not the last symbol of the string must be followed by at least a 1. This gives us the expression $1^*(01^+)^*1^*(\epsilon + 0)$. It is not hard to see that the second 1^* is redundant, and thus the expression can in fact be simplified to $1^*(01^+)^*(\epsilon + 0)$.

Regular expressions were first introduced in Kleene [1956] for studying the properties of neural nets. The preceding examples illustrate that regular expressions often give very clear and concise representations of languages. Unfortunately, not every language can be represented by regular expressions. For example, it will become clear that there is no regular expression for the language $\{0^n1^n \mid n \geq 1\}$. The languages represented by regular expressions are called the **regular languages**. Later, we will see that regular languages are exactly the class of languages generated by the so-called **right-linear grammars**. This connection allows one to prove some interesting mathematical properties about regular languages as well as to design an efficient algorithm to determine whether a given string belongs to the language represented by a given **regular expression**.

Another way of representing languages is to use *pattern systems* [Angluin 1980, Jiang et al. 1995].

Definition 6.9 A *pattern system* is a triple (Σ, V, p), where Σ is the alphabet, V is the set of *variables* with $\Sigma \cap V = \emptyset$, and p is a string over $\Sigma \cup V$ called the *pattern*.

An example pattern system is $(\{0, 1\}, \{v_1, v_2\}, v_1 v_1 0 v_2)$.

Definition 6.10 The language generated by a pattern system (Σ, V, p) consists of all strings over Σ that can be obtained from p by replacing each variable in p with a string over Σ.

For example, the language generated by $(\{0, 1\}, \{v_1, v_2\}, v_1 v_1 0 v_2)$ contains words $0, 00, 01, 000, 001, 010, 011, 110$, etc., but does not contain strings, $1, 10, 11, 100, 101$, etc. The pattern system $(\{0, 1\}, \{v_1\}, v_1 v_1)$ generates the set of all strings, which is the concatenation of two equal substrings, that is, the set $\{xx \mid x \in \{0, 1\}^*\}$. The languages generated by pattern systems are called the *pattern languages*.

Regular languages and pattern languages are really different. One can prove that the pattern language $\{xx \mid x \in \{0, 1\}^*\}$ is not a regular language and the set represented by the regular expression 0^*1^* is not a pattern language. Although it is easy to write an algorithm to decide if a string is in the language generated by a given pattern system, such an algorithm most likely would have to be very inefficient [Angluin 1980].

Perhaps the most useful and general system for representing languages is based on grammars, which are extensions of the pattern systems.

Definition 6.11 A grammar is a quadruple (Σ, N, S, P), where:

1. Σ is a finite nonempty set called the alphabet. The elements of Σ are called the *terminals*.
2. N is a finite nonempty set disjoint from Σ. The elements of N are called the *nonterminals* or *variables*.
3. $S \in N$ is a distinguished nonterminal called the *start symbol*.
4. P is a finite set of *productions* (or *rules*) of the form

$$\alpha \rightarrow \beta$$

where $\alpha \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $\beta \in (\Sigma \cup N)^*$, that is, α is a string of terminals and nonterminals containing at least one nonterminal and β is a string of terminals and nonterminals.

Example 6.7

Let $G_1 = (\{0, 1\}, \{S, T, O, I\}, S, P)$, where P contains the following productions:

$$\begin{aligned} S &\rightarrow OT \\ S &\rightarrow OI \\ T &\rightarrow SI \\ O &\rightarrow 0 \\ I &\rightarrow 1 \end{aligned}$$

As we shall see, the grammar G_1 can be used to describe the set $\{0^n 1^n \mid n \geq 1\}$.

Example 6.8

Let $G_2 = (\{0, 1, 2\}, \{S, A\}, S, P)$, where P contains the following productions.

$$\begin{aligned} S &\rightarrow 0SA2 \\ S &\rightarrow \epsilon \\ 2A &\rightarrow A2 \\ 0A &\rightarrow 01 \\ 1A &\rightarrow 11 \end{aligned}$$

This grammar G_2 can be used to describe the set $\{0^n 1^n 2^n \geq n \geq 0\}$.

Example 6.9

To construct a grammar G_3 to describe English sentences, the alphabet Σ contains all words in English. N would contain nonterminals, which correspond to the structural components in an English sentence, for example, $\langle$sentence$\rangle$, $\langle$subject$\rangle$, $\langle$predicate$\rangle$, $\langle$noun$\rangle$, $\langle$verb$\rangle$, $\langle$article$\rangle$, etc. The start symbol would be $\langle$sentence$\rangle$. Some typical productions are

$$\begin{aligned}
\langle\text{sentence}\rangle &\to \langle\text{subject}\rangle\langle\text{predicate}\rangle\\
\langle\text{subject}\rangle &\to \langle\text{noun}\rangle\\
\langle\text{predicate}\rangle &\to \langle\text{verb}\rangle\langle\text{article}\rangle\langle\text{noun}\rangle\\
\langle\text{noun}\rangle &\to \text{mary}\\
\langle\text{noun}\rangle &\to \text{algorithm}\\
\langle\text{verb}\rangle &\to \text{wrote}\\
\langle\text{article}\rangle &\to \text{an}
\end{aligned}$$

The rule $\langle$sentence$\rangle \to \langle$subject$\rangle\langle$predicate$\rangle$ follows from the fact that a sentence consists of a subject phrase and a predicate phrase. The rules $\langle$noun$\rangle \to$ mary and $\langle$noun$\rangle \to$ algorithm mean that both mary and algorithms are possible nouns.

To explain how a grammar represents a language, we need the following concepts.

Definition 6.12 Let (Σ, N, S, P) be a grammar. A *sentential form* of G is any string of terminals and nonterminals, that is, a string over $\Sigma \cup N$.

Definition 6.13 Let (Σ, N, S, P) be a grammar and γ_1 and γ_2 two sentential forms of G. We say that γ_1 *directly derives* γ_2, denoted $\gamma_1 \Rightarrow \gamma_2$, if $\gamma_1 = \sigma\alpha\tau$, $\gamma_2 = \sigma\beta\tau$, and $\alpha \to \beta$ is a production in P.

For example, the sentential form $00S11$ directly derives the sentential form $00OT11$ in grammar G_1, and $A2A2$ directly derives $AA22$ in grammar G_2.

Definition 6.14 Let γ_1 and γ_2 be two sentential forms of a grammar G. We say that γ_1 *derives* γ_2, denoted $\gamma_1 \Rightarrow^* \gamma_2$, if there exists a sequence of (zero or more) sentential forms $\sigma_1, \ldots, \sigma_n$ such that

$$\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2$$

The sequence $\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2$ is called a derivation from γ_1 to γ_2.

For example, in grammar G_1, $S \Rightarrow^* 0011$ because

$$S \Rightarrow \underline{O}T \Rightarrow 0\underline{T} \Rightarrow 0S\underline{I} \Rightarrow 0\underline{S}1 \Rightarrow 0\underline{O}I1 \Rightarrow 00\underline{I}1 \Rightarrow 0011$$

and in grammar G_2, $S \Rightarrow^* 001122$ because

$$S \Rightarrow 0\underline{S}A2 \Rightarrow 00\underline{S}A2A2 \Rightarrow 0\underline{0A}2A2 \Rightarrow 001\underline{2A}2 \Rightarrow 001\underline{1A}22 \Rightarrow 001122$$

Here the left-hand side of the relevant production in each derivation step is underlined for clarity.

Definition 6.15 Let (Σ, N, S, P) be a grammar. The language generated by G, denoted $L(G)$, is defined as

$$L(G) = \{x \mid x \in \Sigma^*, S \Rightarrow^* x\}$$

The words in $L(G)$ are also called the *sentences* of $L(G)$.

Clearly, $L(G_1)$ contains all strings of the form 0^n1^n, $n \geq 1$, and $L(G_2)$ contains all strings of the form $0^n1^n2^n$, $n \geq 0$. Although only a partial definition of G_3 is given, we know that $L(G_3)$ contains sentences such as "mary wrote an algorithm" and "algorithm wrote an algorithm" but does not contain sentences such as "an wrote algorithm."

The introduction of formal grammars dates back to the 1940s [Post 1943], although the study of rigorous description of languages by grammars did not begin until the 1950s [Chomsky 1956]. In the next subsection, we consider various restrictions on the form of productions in a grammar and see how these restrictions can affect the power of a grammar in representing languages. In particular, we will know that regular languages and pattern languages can all be generated by grammars under different restrictions.

6.4.2 Hierarchy of Grammars

Grammars can be divided into four classes by gradually increasing the restrictions on the form of the productions. Such a classification is due to Chomsky [1956, 1963] and is called the *Chomsky hierarchy*.

Definition 6.16 Let $G = (\Sigma, N, S, P)$ be a grammar.

1. G is also called a *type*-0 *grammar* or an *unrestricted grammar*.
2. G is *type*-1 or **context sensitive** if each production $\alpha \to \beta$ in P either has the form $S \to \epsilon$ or satisfies $|\alpha| \leq |\beta|$.
3. G is *type*-2 or **context free** if each production $\alpha \to \beta$ in P satisfies $|\alpha| = 1$, that is, α is a nonterminal.
4. G is *type*-3 or right linear or regular if each production has one of the following three forms:

$$A \to aB, \qquad A \to a, \qquad A \to \epsilon$$

where A and B are nonterminals and a is a terminal.

The language generated by a type-i is called a type-i language, $i = 0, 1, 2, 3$. A type-1 language is also called a **context-sensitive language** and a type-2 language is also called a **context-free language**. It turns out that every type-3 language is in fact a regular language, that is, it is represented by some regular expression, and vice versa. See the next section for the proof of the equivalence of type-3 (right-linear) grammars and regular expressions.

The grammars G_1 and G_3 given in the last subsection are context free and the grammar G_2 is context sensitive. Now we give some examples of unrestricted and right-linear grammars.

Example 6.10

Let $G_4 = (\{0, 1\}, \{S, A, O, I, T\}, S, P)$, where P contains

$$\begin{array}{rlrl} S & \to AT & & \\ A & \to 0AO & A & \to 1AI \\ O0 & \to 0O & O1 & \to 1O \\ I0 & \to 0I & I1 & \to 1I \\ OT & \to 0T & IT & \to 1T \\ A & \to \epsilon & T & \to \epsilon \end{array}$$

Then G_4 generates the set $\{xx \mid x \in \{0, 1\}^*\}$. For example, we can derive the word 0101 from S as follows:

$$S \Rightarrow \underline{A}T \Rightarrow 0\underline{A}OT \Rightarrow 01\underline{A}IOT \Rightarrow 01I\underline{OT} \Rightarrow 01\underline{I0}T \Rightarrow 010\underline{IT} \Rightarrow 0101\underline{T} \Rightarrow 0101$$

Example 6.11

We give a right-linear grammar G_5 to generate the language represented by the regular expression in Example 6.3, that is, the set of all nonempty binary strings beginning and ending with the same bit. Let $G_5 = (\{0, 1\}, \{S, O, I\}, S, P)$, where P contains

$$\begin{array}{cc} S \to 0O & S \to 1I \\ S \to 0 & S \to 1 \\ O \to 0O & O \to 1O \\ I \to 0I & I \to 1I \\ O \to 0 & I \to 1 \end{array}$$

The following theorem is due to Chomsky [1956, 1963].

Theorem 6.2 *For each $i = 0, 1, 2$, the class of type-i languages properly contains the class of type-$(i + 1)$ languages.*

For example, one can prove by using a technique called *pumping* that the set $\{0^n 1^n \mid n \geq 1\}$ is context free but not regular, and the sets $\{0^n 1^n 2^n \mid n \geq 0\}$ and $\{xx \mid x \in \{0, 1\}^*\}$ are context sensitive but not context free [Hopcroft and Ullman 1979]. It is, however, a bit involved to construct a language that is of type-0 but not context sensitive. See, for example, Hopcroft and Ullman [1979] for such a language.

The four classes of languages in the Chomsky hierarchy also have been completely characterized in terms of Turing machines and their restricted versions. We have already defined a Turing machine in Section 6.2. Many restricted versions of it will be defined in the next section. It is known that type-0 languages are exactly those recognized by Turing machines, context-sensitive languages are those recognized by Turing machines running in linear space, context-free languages are those recognized by Turing machines whose worktapes operate as pushdown stacks [called **pushdown automata** (PDA)], and regular languages are those recognized by Turing machines without any worktapes (called **finite-state machine** or **finite automata**) [Hopcroft and Ullman 1979].

Remark 6.1 Recall our definition of a Turing machine and the function it computes from Section 6.2. In the preceding paragraph, we refer to *a language recognized* by a Turing machine. These are two seemingly different ideas, but they are essentially the same. The reason is that the function f, which maps the set of strings over a finite alphabet to $\{0, 1\}$, corresponds in a natural way to the language L_f over Σ defined as: $L_f = \{x \mid f(x) = 1\}$. Instead of saying that a Turing machine computes the function f, we say equivalently that it recognizes L_f.

Because $\{xx \mid x \in \{0, 1\}^*\}$ is a pattern language, the preceding discussion implies that the class of pattern languages is not contained in the class of context-free languages. The next theorem shows that the class of pattern languages is contained in the class of context-sensitive languages.

Theorem 6.3 *Every pattern language is context sensitive.*

The theorem follows from the fact that every pattern language is recognized by a Turing machine in linear space [Angluin 1980] and linear space-bounded Turing machines recognize exactly context-sensitive languages. To show the basic idea involved, let us construct a context-sensitive grammar for the pattern language $\{xx \mid x \in \{0, 1\}^*\}$. The grammar G_4 given in Example 6.10 for this language is almost context-sensitive. We just have to get rid of the two ϵ-productions: $A \to \epsilon$ and $T \to \epsilon$. A careful modification of G_4 results in the following grammar $G_6 = (\{0, 1\}, \{S, A_0, A_1, O, I, T_0, T_1\}, S, P)$,

where P contains

$$\begin{array}{rlrl} S &\to \epsilon & & \\ S &\to A_0 T_0 & S &\to A_1 T_1 \\ A_0 &\to 0 A_0 O & A_0 &\to 1 A_0 I \\ A_1 &\to 0 A_1 O & A_1 &\to 1 A_1 I \\ A_0 &\to 0 & A_1 &\to 1 \\ O0 &\to 0O & O1 &\to 1O \\ I0 &\to 0I & I1 &\to 1I \\ O T_0 &\to 0 T_0 & I T_0 &\to 1 T_0 \\ O T_1 &\to 0 T_1 & I T_1 &\to 1 T_1 \\ T_0 &\to O & T_1 &\to 1, \end{array}$$

which is context sensitive and generates $\{xx \mid x \in \{0,1\}^*\}$. For example, we can derive 011011 as

$$\Rightarrow \underline{A_1} T_1 \Rightarrow 0\underline{A_1} O T_1 \Rightarrow 01\underline{A_1} I O T_1$$
$$\Rightarrow 011 I \underline{O T_1} \Rightarrow 011 \underline{I0} T_1 \Rightarrow 0110 \underline{I T_1} \Rightarrow 01101 \underline{T_1} \Rightarrow 011011$$

For a class of languages, we are often interested in the so-called *closure properties* of the class.

Definition 6.17 A class of languages (e.g., regular languages) is said to be *closed* under a particular operation (e.g., union, intersection, complementation, concatenation, Kleene closure) if each application of the operation on language(s) of the class results in a language of the class.

These properties are often useful in constructing new languages from existing languages as well as proving many theoretical properties of languages and grammars. The closure properties of the four types of languages in the Chomsky hierarchy are now summarized [Harrison 1978, Hopcroft and Ullman 1979, Gurari 1989].

Theorem 6.4

1. *The class of type-*0 *languages is closed under union, intersection, concatenation, and Kleene closure but not under complementation.*
2. *The class of context-free languages is closed under union, concatenation, and Kleene closure but not under intersection or complementation.*
3. *The classes of context-sensitive and regular languages are closed under all five of the operations.*

For example, let $L_1 = \{0^m 1^n 2^p \mid m = n \text{ or } n = p\}$, $L_2 = \{0^m 1^n 2^p \mid m = n\}$, and $L_3 = \{0^m 1^n 2^p \mid n = p\}$. It is easy to see that all three are context-free languages. (In fact, $L_1 = L_2 \cup L_3$.) However, intersecting L_2 with L_3 gives the set $\{0^m 1^n 2^p \mid m = n = p\}$, which is not context free.

We will look at context-free grammars more closely in the next subsection and introduce the concept of **parsing** and ambiguity.

6.4.3 Context-Free Grammars and Parsing

From a practical point of view, for each grammar $G = (\Sigma, N, S, P)$ representing some language, the following two problems are important:

1. (Membership) Given a string over Σ, does it belong to $L(G)$?
2. (Parsing) Given a string in $L(G)$, how can it be derived from S?

The importance of the membership problem is quite obvious: given an English sentence or computer program we wish to know if it is grammatically correct or has the right format. Parsing is important because a derivation usually allows us to interpret the meaning of the string. For example, in the case of a Pascal program, a derivation of the program in Pascal grammar tells the compiler how the program should be executed. The following theorem illustrates the decidability of the membership problem for the four classes of grammars in the Chomsky hierarchy. The proofs can be found in Chomsky [1963], Harrison [1978], and Hopcroft and Ullman [1979].

Theorem 6.5 *The membership problem for type-*0 *grammars is undecidable in general and is decidable for any context-sensitive grammar* (*and thus for any context-free or right-linear grammars*).

Because context-free grammars play a very important role in describing computer programming languages, we discuss the membership and parsing problems for context-free grammars in more detail. First, let us look at another example of context-free grammar. For convenience, let us abbreviate a set of productions with the same left-hand side nonterminal

$$A \rightarrow \alpha_1, \ldots, A \rightarrow \alpha_n$$

as

$$A \rightarrow \alpha_1 \mid \cdots \mid \alpha_n$$

Example 6.12

We construct a context-free grammar for the set of all valid Pascal real values. In general, a real constant in Pascal has one of the following forms:

$$m.n, \qquad m\mathbf{e}q, \qquad m.n\mathbf{e}q,$$

where m and q are signed or unsigned integers and n is an unsigned integer. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathbf{e}, +, -, .\}$, $N = \{S, M, N, D\}$, and the set P of the productions contain

$$\begin{aligned} S &\rightarrow M.N|M\mathbf{e}M|M.N\mathbf{e}M \\ M &\rightarrow N| + N| - N \\ N &\rightarrow DN|D \\ D &\rightarrow 0|1|2|3|4|5|7|8|9 \end{aligned}$$

Then the grammar generates all valid Pascal real values (including some absurd ones like 001.200**e**000). The value $12.3\mathbf{e} - 4$ can be derived as

$$S \Rightarrow \underline{M}.N\mathbf{e}M \Rightarrow \underline{N}.N\mathbf{e}M \Rightarrow \underline{D}N.N\mathbf{e}M \Rightarrow 1\underline{N}.N\mathbf{e}M \Rightarrow 1\underline{D}.N\mathbf{e}M$$

$$\Rightarrow 12.\underline{N}\mathbf{e}M \Rightarrow 12.\underline{D}\mathbf{e}M \Rightarrow 12.3\mathbf{e}\underline{M} \Rightarrow 12.3\mathbf{e} - \underline{N} \Rightarrow 12.3\mathbf{e} - \underline{D} \Rightarrow 12.3\mathbf{e} - 4$$

Perhaps the most natural representation of derivations for a context-free grammar is *a derivation tree* or *a parse tree*. Each *internal node* of such a tree corresponds to a nonterminal and each *leaf* corresponds to a terminal. If A is an internal node with children $B_1, \ldots, B_n$ ordered from left to right, then $A \rightarrow B_1 \cdots B_n$ must be a production. The concatenation of all leaves from left to right yields the string being derived. For example, the derivation tree corresponding to the preceding derivation of $12.3\mathbf{e} - 4$ is given in Figure 6.3. Such a tree also makes possible the extraction of the parts 12, 3, and −4, which are useful in the storage of the real value in a computer memory.

Definition 6.18 A context-free grammar G is **ambiguous** if there is a string $x \in L(G)$, which has two distinct derivation trees. Otherwise G is *unambiguous*.

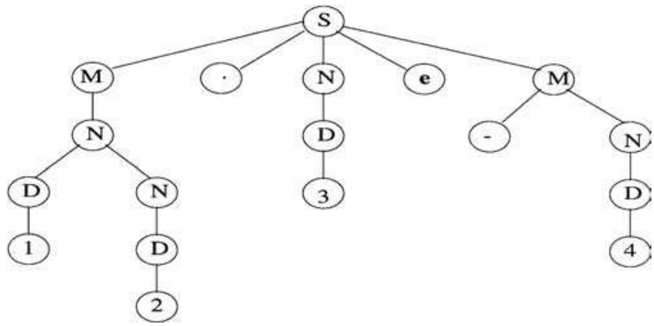


FIGURE 6.3 The derivation tree for 12.3**e** − 4.

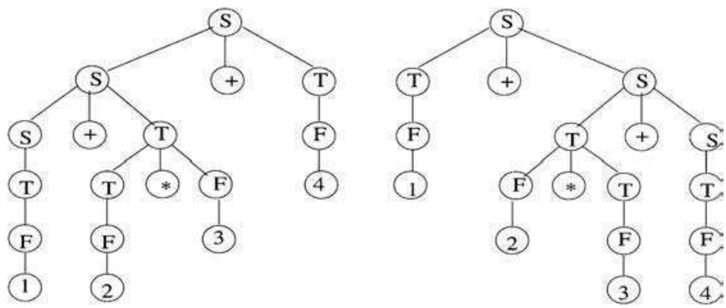


FIGURE 6.4 Different derivation trees for the expression $1 + 2 * 3 + 4$.

Unambiguity is a very desirable property to have as it allows a unique interpretation of each sentence in the language. It is not hard to see that the preceding grammar for Pascal real values and the grammar G_1 defined in Example 6.7 are all unambiguous. The following example shows an ambiguous grammar.

Example 6.13

Consider a grammar G_7 for all valid arithmetic expressions that are composed of unsigned positive integers and symbols $+, *, (,)$. For convenience, let us use the symbol n to denote any unsigned positive integer. This grammar has the productions

$$S \to T + S \mid S + T \mid T$$
$$T \to F * T \mid T * F \mid F$$
$$F \to n \mid (S)$$

Two possible different derivation trees for the expression $1 + 2 * 3 + 4$ are shown in Figure 6.4. Thus, G_7 is ambiguous. The left tree means that the first addition should be done before the second addition and the right tree says the opposite.

Although in the preceding example different derivations/interpretations of any expression always result in the same value because the operations addition and multiplication are associative, there are situations where the difference in the derivation can affect the final outcome. Actually, the grammar G_7 can be made unambiguous by removing some (redundant) productions, for example, $S \to T + S$ and $T \to F * T$. This corresponds to the convention that a sequence of consecutive additions (or multiplications) is always evaluated from left to right and will not change the language generated by G_7. It is worth noting that

there are context-free languages which cannot be generated by any unambiguous context-free grammar [Hopcroft and Ullman 1979]. Such languages are said to be *inherently ambiguous*. An example of inherently ambiguous languages is the set

$$\{0^m 1^m 2^n 3^n \mid m, n > 0\} \cup \{0^m 1^n 2^m 3^n \mid m, n > 0\}$$

We end this section by presenting an efficient algorithm for the membership problem for context-free grammars. The algorithm is due to Cocke, Younger, and Kasami [Hopcroft and Ullman 1979] and is often called the CYK algorithm. Let $G = (\Sigma, N, S, P)$ be a context-free grammar. For simplicity, let us assume that G does not generate the empty string ϵ and that G is in the so-called **Chomsky normal form** [Chomsky 1963], that is, every production of G is either in the form $A \rightarrow BC$ where B and C are nonterminals, or in the form $A \rightarrow a$ where a is a terminal. An example of such a grammar is G_1 given in Example 6.7. This is not a restrictive assumption because there is a simple algorithm which can convert every context-free grammar that does not generate ϵ into one in the Chomsky normal form.

Suppose that $x = a_1 \cdots a_n$ is a string of n terminals. The basic idea of the CYK algorithm, which decides if $x \in L(G)$, is *dynamic programming*. For each pair i, j, where $1 \leq i \leq j \leq n$, define a set $X_{i,j} \subseteq N$ as

$$X_{i,j} = \{A \mid A \Rightarrow^* a_i \cdots a_j\}$$

Thus, $x \in L(G)$ if and only if $S \in X_{1,n}$. The sets $X_{i,j}$ can be computed inductively in the ascending order of $j - i$. It is easy to figure out $X_{i,i}$ for each i because $X_{i,i} = \{A \mid A \rightarrow a_i \in P\}$. Suppose that we have computed all $X_{i,j}$ where $j - i < d$ for some $d > 0$. To compute a set $X_{i,j}$, where $j - i = d$, we just have to find all of the nonterminals A such that there exist some nonterminals B and C satisfying $A \rightarrow BC \in P$ and for some k, $i \leq k < j$, $B \in X_{i,k}$, and $C \in X_{k+1,j}$. A rigorous description of the algorithm in a Pascal style pseudocode is given as follows.

Algorithm CYK($x = a_1 \cdots a_n$):

1. for $i \leftarrow 1$ to n do
2. $\quad X_{i,i} \leftarrow \{A \mid A \rightarrow a_i \in P\}$
3. for $d \leftarrow 1$ to $n - 1$ do
4. $\quad$ for $i \leftarrow 1$ to $n - d$ do
5. $\quad X_{i,i+d} \leftarrow \emptyset$
6. $\quad$ for $t \leftarrow 0$ to $d - 1$ do
7. $\qquad X_{i,i+d} \leftarrow X_{i,i+d} \cup \{A \mid A \rightarrow BC \in P$ for some $B \in X_{i,i+t}$ and $C \in X_{i+t+1,i+d}\}$

Table 6.2 shows the sets $X_{i,j}$ for the grammar G_1 and the string $x = 000111$. It just so happens that every $X_{i,j}$ is either empty or a singleton. The computation proceeds from the main diagonal toward the upper-right corner.

TABLE 6.2 An Example Execution of the CYK Algorithm

		0	0	0	1	1	1
		$j \rightarrow$					
		1	2	3	4	5	6
	1	O					S
	2		O			S	T
i	3			O	S	T	
$\downarrow$	4				I		
	5					I	
	6						I

6.5 Computational Models

In this section, we will present many restricted versions of Turing machines and address the question of what kinds of problems they can solve. Such a classification is a central goal of computation theory. We have already classified problems broadly into (totally) decidable, partially decidable, and totally undecidable. Because the decidable problems are the ones of most practical interest, we can consider further classification of decidable problems by placing two types of restrictions on a Turing machine. The first one is to restrict its structure. This way we obtain many machines of which a finite automaton and a pushdown automaton are the most important. The other way to restrict a Turing machine is to bound the amount of resources it uses, such as the number of time steps or the number of tape cells it can use. The resulting machines form the basis for *complexity theory*.

6.5.1 Finite Automata

The finite automaton (in its deterministic version) was first introduced by McCulloch and Pitts [1943] as a logical model for the behavior of neural systems. Rabin and Scott [1959] introduced the nondeterministic version of the finite automaton and showed the equivalence of the nondeterministic and deterministic versions. Chomsky and Miller [1958] proved that the set of languages that can be recognized by a finite automaton is precisely the regular languages introduced in Section 6.4. Kleene [1956] showed that the languages accepted by finite automata are characterized by regular expressions as defined in Section 6.4.

In addition to their original role in the study of neural nets, finite automata have enjoyed great success in many fields such as sequential circuit analysis in circuit design [Kohavi 1978], asynchronous circuits [Brzozowski and Seger 1994], lexical analysis in text processing [Lesk 1975], and compiler design. They also led to the design of more efficient algorithms. One excellent example is the development of linear-time string-matching algorithms, as described in Knuth et al. [1977]. Other applications of finite automata can be found in computational biology [Searls 1993], natural language processing, and distributed computing.

A finite automaton, as in Figure 6.5, consists of an input tape which contains a (finite) sequence of input symbols such as $aabab\cdots$, as shown in the figure, and a finite-state control. The tape is read by the one-way *read-only* input head from left to right, one symbol at a time. Each time the input head reads an input symbol, the finite control changes its state according to the symbol and the current state of the machine. When the input head reaches the right end of the input tape, if the machine is in a final state, we say that the input is accepted; if the machine is not in a final state, we say that the input is rejected. The following is the formal definition.

Definition 6.19 A *nondeterministic finite automaton* (NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of *states*.
- Σ is a finite set of *input symbols*.
- δ, the *state transition function*, is a mapping from $Q \times \Sigma$ to subsets of Q.
- $q_0 \in Q$ is the *initial state* of the NFA.
- $F \subseteq Q$ is the set of *final states*.

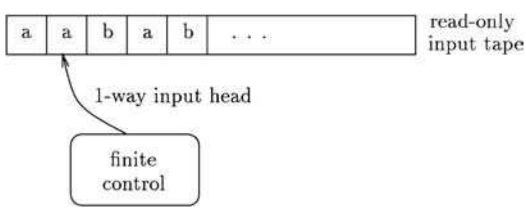


FIGURE 6.5 A finite automaton.

If δ maps $|Q| \times \Sigma$ to singleton subsets of Q, then we call such a machine a *deterministic finite automaton* (DFA).

When an automaton, M, is nondeterministic, then from the current state and input symbol, it may go to one of several different states. One may imagine that the device goes to all such states in parallel. The DFA is just a special case of the NFA; it always follows a single deterministic path. The device M *accepts* an input string x if, starting with q_0 and the read head at the first symbol of x, one of these parallel paths reaches an accepting state when the read head reaches the end of x. Otherwise, we say M *rejects* x. A language, L, is accepted by M if M accepts all of the strings in L and nothing else, and we write $L = L(M)$. We will also allow the machine to make ϵ-*transitions*, that is, changing state without advancing the read head. This allows transition functions such as $\delta(s, \epsilon) = \{s'\}$. It is easy to show that such a generalization does not add more power.

Remark 6.2 The concept of a nondeterministic automaton is rather confusing for a beginner. But there is a simple way to relate it to a concept which must be familiar to all of the readers. It is that of a solitaire game. Imagine a game like *Klondike*. The game starts with a certain arrangement of cards (the input) and there is a well-defined final position that results in success; there are also dead ends where a further move is not possible; you lose if you reach any of them. At each step, the precise rules of the game dictate how a new arrangement of cards can be reached from the current one. But the most important point is that there are many possible moves at each step. (Otherwise, the game would be no fun!) Now consider the following question: What starting positions are *winnable*? These are the starting positions for which *there is a winning move sequence*; of course, in a typical play a player may not achieve it. But that is beside the point in the definition of what starting positions are winnable. The connection between such games and a nondeterministic automaton should be clear. The multiple choices at each step are what make it *nondeterministic*. Our definition of winnable positions is similar to the concept of acceptance of a string by a nondeterministic automaton. Thus, an NFA may be viewed as a formal model to define solitaire games.

Example 6.14

We design a DFA to accept the language represented by the regular expression $0(0+1)^*1$ as in Example 6.2, that is, the set of all strings in $\{0, 1\}$ which begin with a 0 and end with a 1. It is usually convenient to draw our solution as in Figure 6.6. As a convention, each circle represents a state; the state a, pointed at by the initial arrow, is the initial state. The darker circle represents the final states (state c). The transition function δ is represented by the labeled edges. For example, $\delta(\mathrm{a}, 0) = \{\mathrm{b}\}$. When a transition is missing, for example on input 1 from a and on inputs 0 and 1 from c, it is assumed that all of these lead to an implicit nonaccepting trap state, which has transitions to itself on all inputs.

The machine in Figure 6.6 is nondeterministic because from b on input 1 the machine has two choices: stay at b or go to c.

Figure 6.7 gives an equivalent DFA, accepting the same language.

Example 6.15

The DFA in Figure 6.8 accepts the set of all strings in $\{0, 1\}^*$ with an even number of 1s. The corresponding regular expression is $(0^*10^*1)^*0^*$.

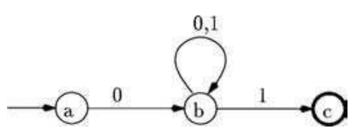


FIGURE 6.6 An NFA accepting $0(0+1)^*1$.

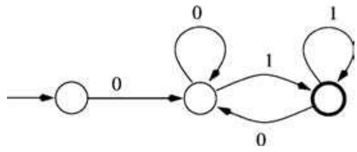


FIGURE 6.7 A DFA accepting $0(0+1)^*1$.

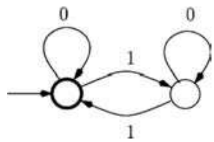


FIGURE 6.8 A DFA accepting $(0^*10^*1)^*0^*$.

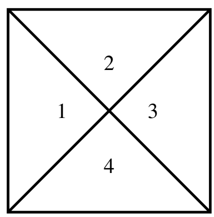


FIGURE 6.9 Numbering the quarters of a tile.

Example 6.16

As a final example, consider the special case of the tiling problem that we discussed in Section 6.2. This version of the problem is as follows: Let k be a fixed positive integer. Given a set of unit tiles, we want to know if they can tile any $k \times n$ area for all n. We show how to deal with the case $k = 1$ and leave it as an exercise to generalize our method for larger values of k. Number the quarters of each tile as in Figure 6.9. The given set of tiles will tile the area if we can find a sequence of the given tiles $T_1, T_2, \ldots, T_m$ such that (1) the third quarter of T_1 has the same color as the first quarter of T_2, and the third quarter of T_2 has the same color as the first quarter of T_3, etc., and (2) the third quarter of T_m has the same color as T_1. These conditions can be easily understood as follows. The first condition states that the tiles T_1, T_2, etc., can be placed adjacent to each other along a row in that order. The second condition implies that the whole sequence $T_1\ T_2 \cdots T_m$ can be replicated any number of times. And a little thought reveals that this is all we need to answer yes on the input. But if we cannot find such a sequence, then the answer must be no. Also note that in the sequence no tile needs to be repeated and so the value of m is bounded by the number of tiles in the input. Thus, we have reduced the problem to searching a finite number of possibilities and we are done.

How is the preceding discussion related to finite automata? To see the connection, define an alphabet consisting of the unit tiles and define a language $L = \{T_1 T_2 \cdots T_m \mid T_1 T_2 \cdots T_m$ is a valid tiling, $m \geq 0\}$. We will now construct an NFA for the language L. It consists of states corresponding to *distinct* colors contained in the tiles plus two states, one of them the start state and another state called the dead state. The NFA makes transitions as follows: From the start state there is an ϵ-transition to each color state, and all states except the dead state are accepting states. When in the state corresponding to color i, suppose it receives input tile T. If the first quarter of this tile has color i, then it moves to the color of the third quarter of T; otherwise, it enters the dead state. The basic idea is to remember the only relevant piece

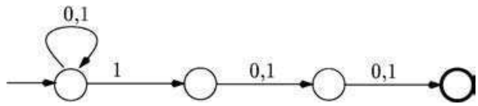


FIGURE 6.10 An NFA accepting L_3.

of information after processing some input. In this case, it is the third quarter color of the last tile seen. Having constructed this NFA, the question we are asking is if the language accepted by this NFA is infinite. There is a simple algorithm for this problem [Hopcroft and Ullman 1979].

The next three theorems show a satisfying result that all the following language classes are identical:

- The class of languages accepted by DFAs
- The class of languages accepted by NFAs
- The class of languages generated by regular expressions, as in Definition 6.8
- The class of languages generated by the right-linear, or type-3, grammars, as in Definition 6.16

Recall that this class of languages is called the *regular languages* (see Section 6.4).

Theorem 6.6 *For each NFA, there is an equivalent DFA.*

Proof An NFA might look more powerful because it can carry out its computation in parallel with its nondeterministic branches. But because we are working with a *finite number* of states, we can simulate an NFA $M = (Q, \Sigma, \delta, q_0, F)$ by a DFA $M' = (Q', \Sigma, \delta', q_0', F')$, where

- $Q' = \{[S] : S \subseteq Q\}$.
- $q_0' = [\{q_0\}]$.
- $\delta'([S], a) = [S'] = [\cup_{q_l \in S} \delta(q_l, a)]$.
- F' is the set of all subsets of Q containing a state in F.

It can now be verified that $L(M) = L(M')$. □

Example 6.17

Example 6.1 contains an NFA and an equivalent DFA accepting the same language. In fact, the proof provides an effective procedure for converting an NFA to a DFA. Although each NFA can be converted to an equivalent DFA, the resulting DFA might be exponentially large in terms of the number of states, as we can see from the previous procedure. This turns out to be the best thing one can do in the worst case. Consider the language: $L_k = \{x : x \in \{0, 1\}^*$ and the kth letter from the right of x is a $1\}$. An NFA of $k + 1$ states (for $k = 3$) accepting L_k is given in Figure 6.10. A counting argument shows that any DFA accepting L_k must have at least 2^k states.

Theorem 6.7 *L is generated by a right-linear grammar if it is accepted by an* NFA.

Proof Let L be accepted by a right-linear grammar $G = (\Sigma, N, S, P)$. We design an NFA $M = (Q, \Sigma, \delta, q_0, F)$ where $Q = N \cup \{f\}$, $q_0 = S$, $F = \{f\}$. To define the δ function, we have $C \in \delta(A, b)$ if $A \to bC$. For rules $A \to b$, $\delta(A, b) = \{f\}$. Obviously, $L(M) = L(G)$.

Conversely, if L is accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$, we define an equivalent right-linear grammar $G = (\Sigma, N, S, P)$, where $N = Q$, $S = q_0$, $q_i \to aq_j \in N$ if $q_j \in \delta(q_i, a)$, and $q_j \to \epsilon$ if $q_j \in F$. Again it is easily seen that $L(M) = L(G)$. □

Theorem 6.8 *L is generated by a regular expression if it is accepted by an* NFA.

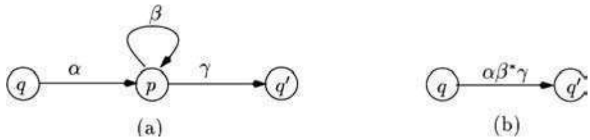


FIGURE 6.11 Converting an NFA to a regular expression.

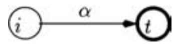


FIGURE 6.12 The reduced NFA.

Proof (Idea) **Part 1.** We inductively convert a regular expression to an NFA which accepts the language generated by the regular expression as follows.

- Regular expression ϵ converts to $(\{q\}, \Sigma, \emptyset, q, \{q\})$.
- Regular expression $\emptyset$ converts to $(\{q\}, \Sigma, \emptyset, q, \emptyset)$.
- Regular expression a, for each $a \in \Sigma$ converts to $(\{q, f\}, \Sigma, \delta(q, a) = \{f\}, q, \{f\})$.
- If α and β are regular expressions, converting to NFAs M_α and M_β, respectively, then the regular expression $\alpha \cup \beta$ converts to an NFA M, which connects M_α and M_β in parallel: M has an initial state q_0 and all of the states and transitions of M_α and M_β; by ϵ-transitions, M goes from q_0 to the initial states of M_α and M_β.
- If α and β are regular expressions, converting to NFAs M_α and M_β, respectively, then the regular expression $\alpha\beta$ converts to NFA M, which connects M_α and M_β sequentially: M has all of the states and transitions of M_α and M_β, with M_α's initial state as M's initial state, ϵ-transition from the final states of M_α to the initial state of M_β, and M_β's final states as M's final states.
- If α is a regular expression, converting to NFA M_α, then connecting all of the final states of M_α to its initial state with ϵ-transitions gives α^+. Union of this with the NFA for ϵ gives the NFA for α^*.

Part 2. We now show how to convert an NFA to an equivalent regular expression. The idea used here is based on Brzozowski and McCluskey [1963]; see also Brzozowski and Seger [1994] and Wood [1987].

Given an NFA M, expand it to M' by adding two extra states i, the initial state of M', and t, the only final state of M', with ϵ transitions from i to the initial state of M and from all final states of M to t. Clearly, $L(M) = L(M')$. In M', remove states other than i and t one by one as follows. To remove state p, for each triple of states q, p, q' as shown in Figure 6.11a, add the transition as shown in Figure 6.11(b). □

If p does not have a transition leading back to itself, then $\beta = \epsilon$. After we have considered all such triples, delete state p and transitions related to p. Finally, we obtain Figure 6.12 and $L(\alpha) = L(M)$.

Apparently, DFAs cannot serve as our model for a modern computer. Many extremely simple languages cannot be accepted by DFAs. For example, $L = \{xx : x \in \{0, 1\}^*\}$ cannot be accepted by a DFA. One can prove this by counting, or using the so-called pumping lemmas; one can also prove this by arguing that x contains more information than a *finite* state machine can *remember*. We refer the interested readers to textbooks such as Hopcroft and Ullmann [1979], Gurari [1989], Wood [1987], and Floyd and Beigel [1994] for traditional approaches and to Li and Vitányi [1993] for a nontraditional approach. One can try to generalize the DFA to allow the input head to be *two way* but still read only. But such machines are not more powerful, they can be simulated by normal DFAs. The next step is apparently to add *storage* space such that our machines can *write* information in.

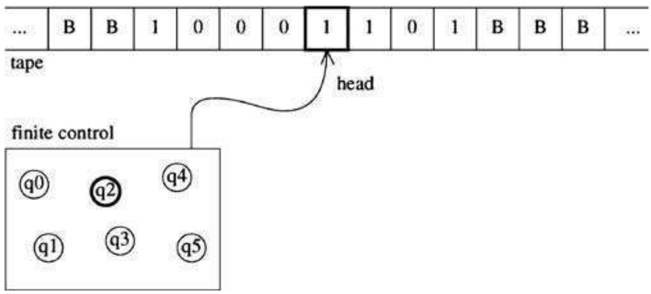


FIGURE 6.13 A Turing machine.

6.5.2 Turing Machines

In this section we will provide an alternative definition of a Turing machine to make it compatible with our definitions of a DFA, PDA, etc. This also makes it easier to define a nondeterministic Turing machine. But this formulation (at least the deterministic version) is essentially the same as the one presented in Section 6.2.

A Turing machine (TM), as in Figure 6.13, consists of a *finite control*, an infinite *tape* divided into cells, and a read/write *head* on the tape. We refer to the two directions on the tape as *left* and *right*. The finite control can be in any one of a finite set Q of states, and each tape cell can contain a 0, a 1, or a *blank* B. Time is discrete and the time instants are ordered $0, 1, 2, \ldots$ with 0 the time at which the machine starts its computation. At any time, the head is positioned over a particular cell, which it is said to *scan*. At time 0 the head is situated on a distinguished cell on the tape called the *start cell*, and the finite control is in the initial state q_0. At time 0 all cells contain Bs, except a contiguous finite sequence of cells, extending from the start cell to the right, which contain 0s and 1s. This binary sequence is called the *input*.

The device can perform the following basic operations:

1. It can write an element from the tape alphabet $\Sigma = \{0, 1, B\}$ in the cell it scans.
2. It can shift the head one cell left or right.

Also, the device executes these operations at the rate of one operation per time unit (a *step*). At the conclusion of each step, the finite control takes on a state in Q. The device operates according to a finite set P of *rules*.

The rules have format (p, s, a, q) with the meaning that if the device is in state p and s is the symbol under scan then write a if $a \in \{0, 1, B\}$ or move the head according to a if $a \in \{L, R\}$ and the finite control changes to state q. At some point, if the device gets into a special *final* state q_f, the device stops and accepts the input.

If every pair of distinct quadruples differs in the first two elements, then the device is *deterministic*. Otherwise, the device is *nondeterministic*. Not every possible combination of the first two elements has to be in the set; in this way we permit the device to perform *no* operation. In this case, we say the device *halts*. In this case, if the machine is not in a final state, we say that the machine *rejects* the input.

Definition 6.20 A Turing machine is a quintuple $M = (Q, \Sigma, P, q_0, q_f)$ where each of the components has been described previously.

Given an input, a deterministic Turing machine carries out a uniquely determined succession of operations, which may or may not terminate in a finite number of steps. If it terminates, then the nonblank symbols left on the tape are the output. Given an input, a **nondeterministic Turing machine** behaves much like an NFA. One may imagine that it carries out its computation in parallel. Such a computation may be viewed as a (possibly infinite) tree. The root of the tree is the starting configuration of the machine.

The children of each node are all possible configurations one step away from this node. If any of the branches terminates in the final state q_f, we say the machine accepts the input. The reader may want to test understanding this new formulation of a Turing machine by redoing the doubling program on a Turing machine with states and transitions (rather than a GOTO program).

A Turing machine *accepts* a language L if $L = \{w : M \text{ accepts } w\}$. Furthermore, if M halts on all inputs, then we say that L is *Turing decidable*, or *recursive*. The connection between a recursive language and a decidable problem (function) should be clear. It is that function f is decidable if and only if L_f is recursive. (Readers who may have forgotten the connection between function f and the associated language L_f should review Remark 6.1.)

Theorem 6.9 *All of the following generalizations of Turing machines can be simulated by a one-tape deterministic Turing machine defined in Definition 6.20.*

- *Larger tape alphabet* Σ
- *More work tapes*
- *More access points, read/write heads, on each tape*
- *Two- or more dimensional tapes*
- *Nondeterminism*

Although these generalizations do not make a Turing machine compute more, they do make a Turing machine more efficient and easier to program. Many more variants of Turing machines are studied and used in the literature. Of all simulations in Theorem 6.9, the last one needs some comments. A nondeterministic computation branches like a tree. When simulating such a computation for n steps, the obvious thing for a deterministic Turing machine to do is to try all possibilities; thus, this requires up to c^n steps, where c is the maximum number of nondeterministic choices at each step.

Example 6.18

A DFA is an extremely simple Turing machine. It just reads the input symbols from left to right. Turing machines naturally accept more languages than DFAs can. For example, a Turing machine can accept $L = \{xx : x \in \{0, 1\}^*\}$ as follows:

- Find the middle point first: it is trivial by using two heads; with one head, one can mark one symbol at the left and then mark another on the right, and go back and forth to eventually find the middle point.
- Match the two parts: with two heads, this is again trivial; with one head, one can again use the marking method matching a pair of symbols each round; if the two parts match, accept the input by entering q_f.

There are types of storage media other than a tape:

- A *pushdown store* is a semi-infinite work tape with one head such that each time the head moves to the left, it erases the symbol scanned previously; this is a last-in first-out storage.
- A *queue* is a semi-infinite work tape with two heads that move only to the right, the leading head is write-only and the trailing head is read-only; this is a first-in first-out storage.
- A *counter* is a pushdown store with a single-letter alphabet (except its one end, which holds a special marker symbol). Thus, a counter can store a nonnegative integer and can perform three operations.

A queue machine can simulate a normal Turing machine, but the other two types of machines are not powerful enough to simulate a Turing machine.

Example 6.19

When the Turing machine tape is replaced by a pushdown store, the machine is called a *pushdown automaton*. Pushdown automata have been thoroughly studied because they accept the class of context-free

languages defined in Section 6.4. More precisely, it can be shown that if L is a context-free language, then it is accepted by a PDA, and if L is accepted by a PDA, then there is a CFG generating L. Various types of PDAs have fundamental applications in compiler design.

The PDA is more restricted than a Turing machine. For example, $L = \{xx : x \in \{0, 1\}^*\}$ cannot be accepted by a PDA, but it can be accepted by a Turing machine as in Example 6.18. But a PDA is more powerful than a DFA. For example, a PDA can accept the language $L' = \{0^k 1^k : k \geq 0\}$ easily. It can read the 0s and push them into the pushdown store; then, after it finishes the 0s, each time the PDA reads a 1, it removes a 0 from the pushdown store; at the end, it accepts if the pushdown store is empty (the number of 0s matches that of 1s). But a DFA cannot accept L', because after it has read all of the 0s, it cannot remember k when k has higher information content than the DFA's finite control.

Two pushdown stores can be used to simulate a tape easily. For comparisons of powers of pushdown stores, queues, counters, and tapes, see van Emde Boas [1990] and Li and Vitányi [1993].

The idea of the universal algorithm was introduced in Section 6.2. Formally, a *universal Turing machine*, U, takes an encoding of a pair of parameters (M, x) as input and simulates M on input x. U accepts (M, x) iff M accepts x. The universal Turing machines have many applications. For example, the definition of Kolmogorov complexity [Li and Vitányi 1993] fundamentally relies on them.

Example 6.20

Let $L_u = \{\langle M, w\rangle : M \text{ accepts } w\}$. Then L_u can be accepted by a Turing machine, but it is not Turing decidable. The proof is omitted.

If a language is Turing acceptable but not Turing decidable, we call such a language *recursively enumerable* (r.e.). Thus, L_u is r.e. but not recursive. It is easily seen that if both a language and its complement are r.e., then both of them are recursive. Thus, $\bar{L}_u$ is not r.e.

6.5.2.1 Time and Space Complexity

With Turing machines, we can now formally define what we mean by **time and space complexities**. Such a formal investigation by Hartmanis and Stearns [1965] marked the beginning of the field of *computational complexity*. We refer the readers to Hartmanis' Turing Award lecture [Hartmanis 1994] for an interesting account of the history and the future of this field.

To define the space complexity properly (in the sublinear case), we need to slightly modify the Turing machine of Figure 6.13. We will replace the tape containing the input by a read-only input tape and give the Turing machine some extra work tapes.

Definition 6.21 Let M be a Turing machine. If for each n, for each input of length n, and for each sequence of choices of moves when M is nondeterministic, M makes at most $T(n)$ moves we say that M is of *time complexity* $T(n)$; similarly, if M uses at most $S(n)$ tape cells of the work tape, we say that M is of space complexity $S(n)$.

Theorem 6.10 *Any Turing machine using $s(n)$ space can be simulated by a Turing machine, with just one work tape, using $s(n)$ space. If a language is accepted by a k-tape Turing machine running in time $t(n)$ [space $s(n)$], then it also can be accepted by another k-tape Turing machine running in time $ct(n)$ [space $cs(n)$], for any constant $c > 0$.*

To avoid writing the constant c everywhere, we use the standard big-O notation: we say $f(n)$ is $O(g(n))$ if there is a constant c such that $f(n) \leq cg(n)$ for all but finitely many n. The preceding theorem is called the linear speedup theorem; it can be proved easily by using a larger tape alphabet to encode several cells into one and hence compress several steps into one. It leads to the following definitions.

Definition 6.22

DTIME$[t(n)]$ is the set of languages accepted by multitape deterministic TMs in time $O(t(n))$.

NTIME$[t(n)]$ is the set of languages accepted by multitape nondeterministic TMs in time $O(t(n))$.

DSPACE[$s(n)$] is the set of languages accepted by multitape deterministic TMs in space $O(s(n))$.
NSPACE[$s(n)$] is the set of languages accepted by multitape nondeterministic TMs in space $O(s(n))$.
P is the complexity class $\bigcup_{c \in \mathcal{N}}$ DTIME[n^c].
NP is the complexity class $\bigcup_{c \in \mathcal{N}}$ NTIME[n^c].
PSPACE is the complexity class $\bigcup_{c \in \mathcal{N}}$ DSPACE[n^c].

Example 6.21

We mentioned in Example 6.18 that $L = \{xx : x \in \{0, 1\}^*\}$ can be accepted by a Turing machine. The procedure we have presented in Example 6.18 for a one-head one-tape Turing machine takes $O(n^2)$ time because the single head must go back and forth marking and matching. With two heads, or two tapes, L can be easily accepted in $O(n)$ time.

It should be clear that any language that can be accepted by a DFA, an NFA, or a PDA can be accepted by a Turing machine in $O(n)$ time. The type-1 grammar in Definition 6.16 can be accepted by a Turing machine in $O(n)$ space. Languages in P, that is, languages acceptable by Turing machines in *polynomial* time, are considered as *feasibly* computable. It is important to point out that all generalizations of the Turing machine, except the nondeterministic version, can all be simulated by the basic one-tape deterministic Turing machine with at most polynomial slowdown. The class NP represents the class of languages accepted in polynomial time by a nondeterministic Turing machine. The nondeterministic version of PSPACE turns out to be identical to PSPACE [Savitch 1970]. The following relationships are true:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$$

Whether or not either of the inclusions is proper is one of the most fundamental open questions in computer science and mathematics. Research in computational complexity theory centers around these questions. To solve these problems, one can identify the hardest problems in NP or PSPACE. These topics will be discussed in Chapter 8. We refer the interested reader to Gurari [1989], Hopcroft and Ullman [1979], Wood [1987], and Floyd and Beigel [1994].

6.5.2.2 Other Computing Models

Over the years, many alternative computing models have been proposed. With reasonable complexity measures, they can all be simulated by Turing machines with at most a polynomial slowdown. The reference van Emde Boas [1990] provides a nice survey of various computing models other than Turing machines. Because of limited space, we will discuss a few such alternatives very briefly and refer our readers to van Emde Boas [1990] for details and references.

Random Access Machines. The *random access machine* (RAM) [Cook and Reckhow 1973] consists of a finite control where a program is stored, with several arithmetic registers and an infinite collection of memory registers $R[1], R[2], \ldots$. All registers have an unbounded word length. The basic instructions for the program are LOAD, ADD, MULT, STORE, GOTO, ACCEPT, REJECT, etc. Indirect addressing is also used. Apparently, compared to Turing machines, this is a closer but more complicated approximation of modern computers. There are two standard ways for measuring time complexity of the model:

- The *unit-cost RAM:* in this case, each instruction takes one unit of time, no matter how big the operands are. This measure is convenient for analyzing some algorithms such as sorting. But it is unrealistic or even meaningless for analyzing some other algorithms, such as integer multiplication.
- The *log-cost RAM:* each instruction is charged for the sum of the lengths of all data manipulated implicitly or explicitly by the instruction. This is a more realistic model but sometimes less convenient to use.

Log-cost RAMs and Turing machines can simulate each other with polynomial overheads. The unit-cost RAM might be exponentially (but unrealistically) faster when, for example, it uses its power of multiplying two large numbers in one step.

Pointer Machines. The pointer machines were introduced by Kolmogorov and Uspenskii [1958] (also known as the Kolmogorov–Uspenskii machine) and by Schönhage in 1980 (also known as the storage modification machine, see Schönhage [1980]). We informally describe the pointer machine here. A pointer machine is similar to a RAM but differs in its memory structure. A pointer machine operates on a storage structure called a Δ structure, where Δ is a finite alphabet of size greater than one. A Δ-structure S is a finite directed graph (the Kolmogorov–Uspenskii version is an undirected graph) in which each node has $k = |\Delta|$ outgoing edges, which are labeled by the k symbols in Δ. S has a distinguished node called the *center*, which acts as a starting point for addressing, with words over Δ, other nodes in the structure. The pointer machine has various instructions to redirect the pointers or edges and thus modify the storage structure. It should be clear that Turing machines and pointer machines can simulate each other with at most polynomial delay if we use the log-cost model as with the RAMs. There are many interesting studies on the efficiency of the preceding simulations. We refer the reader to van Emde Boas [1990] for more pointers on the pointer machines.

Circuits and Nonuniform Models. A *Boolean circuit* is a finite, labeled, directed acyclic graph. Input nodes are nodes without ancestors; they are labeled with input variables $x_1, \ldots, x_n$. The internal nodes are labeled with functions from a finite set of Boolean operations, for example, {and, or, not} or $\{\oplus\}$. The number of ancestors of an internal node is precisely the number of arguments of the Boolean function that the node is labeled with. A node without successors is an output node. The circuit is naturally evaluated from input to output: at each node the function labeling the node is evaluated using the results of its ancestors as arguments. Two cost measures for the circuit model are:

- *Depth:* the length of a longest path from an input node to an output node
- *Size:* the number of nodes in the circuit

These measures are applied to a family of circuits $\{C_n : n \geq 1\}$ for a particular problem, where C_n solves the problem of size n. If C_n can be computed from n (in polynomial time), then this is a *uniform measure*. Such circuit families are equivalent to Turing machines. If C_n cannot be computed from n, then such measures are *nonuniform* measures, and such classes of circuits are more powerful than Turing machines because they simply can compute any function by encoding the solutions of all inputs for each n. See van Emde Boas [1990] for more details and pointers to the literature.

Acknowledgment

We would like to thank John Tromp and the reviewers for reading the initial drafts and helping us to improve the presentation.

Defining Terms

Algorithm A finite sequence of instructions that is supposed to solve a particular problem.

Ambiguous context-free grammar For some string of terminals the grammar has two distinct derivation trees.

Chomsky normal form: Every rule of the context-free grammar has the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, and C are nonterminals and a is a terminal.

Computable or decidable function/problem: A function/problem that can be solved by an algorithm (or equivalently, a Turing machine).

Context-free grammar: A grammar whose rules have the form $A \rightarrow \beta$, where A is a nonterminal and β is a string of nonterminals and terminals.

Context-free language: A language that can be described by some context-free grammar.

Context-sensitive grammar: A grammar whose rules have the form $\alpha \rightarrow \beta$, where α and β are strings of nonterminals and terminals and $|\alpha| \leq |\beta|$.

Context-sensitive language: A language that can be described by some context-sensitive grammar.

Derivation or parsing: An illustration of how a string of terminals is obtained from the start symbol by successively applying the rules of the grammar.

Finite automaton or finite-state machine: A restricted Turing machine where the head is read only and shifts only from left to right.

(Formal) grammar: A description of some language typically consisting of a set of terminals, a set of nonterminals with a distinguished one called the start symbol, and a set of rules (or productions) of the form $\alpha \to \beta$, depicting what string α of terminals and nonterminals can be rewritten as another string β of terminals and nonterminals.

(Formal) language: A set of strings over some fixed alphabet.

Halting problem: The problem of deciding if a given program (or Turing machine) halts on a given input.

Nondeterministic Turing machine: A Turing machine that can make any one of a prescribed set of moves on a given state and symbol read on the tape.

Partially decidable decision problem: There exists a program that always halts and outputs 1 for every input expecting a positive answer and either halts and outputs 0 or loops forever for every input expecting a negative answer.

Program: A sequence of instructions that is not required to terminate on every input.

Pushdown automaton: A restricted Turing machine where the tape acts as a pushdown store (or a stack).

Reduction: A computable transformation of one problem into another.

Regular expression: A description of some language using operators union, concatenation, and Kleene closure.

Regular language: A language that can be described by some right-linear/regular grammar (or equivalently by some regular expression).

Right-linear or regular grammar: A grammar whose rules have the form $A \to aB$ or $A \to a$, where A, B are nonterminals and a is either a terminal or the null string.

Time/space complexity: A function describing the maximum time/space required by the machine on any input of length n.

Turing machine: A simplest formal model of computation consisting of a finite-state control and a semi-infinite sequential tape with a read–write head. Depending on the current state and symbol read on the tape, the machine can change its state and move the head to the left or right.

Uncomputable or undecidable function/problem: A function/problem that cannot be solved by any algorithm (or equivalently, any Turing machine).

Universal algorithm: An algorithm that is capable of simulating any other algorithms if properly encoded.

References

Angluin, D. 1980. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.* 21:46–62.

Brzozowski, J. and McCluskey, E., Jr. 1963. Signal flow graph techniques for sequential circuit state diagram. *IEEE Trans. Electron. Comput.* EC-12(2):67–76.

Brzozowski, J. A. and Seger, C.-J. H. 1994. *Asynchronous Circuits*. Springer–Verlag, New York.

Chomsky, N. 1956. Three models for the description of language. *IRE Trans. Inf. Theory* 2(2):113–124.

Chomsky, N. 1963. Formal properties of grammars. In *Handbook of Mathematical Psychology,* Vol. 2, pp. 323–418. John Wiley and Sons, New York.

Chomsky, N. and Miller, G. 1958. Finite-state languages. *Information and Control* 1:91–112.

Cook, S. and Reckhow, R. 1973. Time bounded random access machines. *J. Comput. Syst. Sci.* 7:354–375.

Davis, M. 1980. What is computation? In *Mathematics Today–Twelve Informal Essays*. L. Steen, ed., pp. 241–259. Vintage Books, New York.

Floyd, R. W. and Beigel, R. 1994. *The Language of Machines: An Introduction to Computability and Formal Languages*. Computer Science Press, New York.

Gurari, E. 1989. *An Introduction to the Theory of Computation*. Computer Science Press, Rockville, MD.

Harel, D. 1992. *Algorithmics: The Spirit of Computing*. Addison–Wesley, Reading, MA.

Harrison, M. 1978. *Introduction to Formal Language Theory*. Addison–Wesley, Reading, MA.

Hartmanis, J. 1994. On computational complexity and the nature of computer science. *Commun. ACM* 37(10):37–43.

Hartmanis, J. and Stearns, R. 1965. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.* 117:285–306.

Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison–Wesley, Reading, MA.

Jiang, T., Salomaa, A., Salomaa, K., and Yu, S. 1995. Decision problems for patterns. *J. Comput. Syst. Sci.* 50(1):53–63.

Kleene, S. 1956. Representation of events in nerve nets and finite automata. In *Automata Studies*, pp. 3–41. Princeton University Press, Princeton, NJ.

Knuth, D., Morris, J., and Pratt, V. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6:323–350.

Kohavi, Z. 1978. *Switching and Finite Automata Theory*. McGraw–Hill, New York.

Kolmogorov, A. and Uspenskii, V. 1958. On the definition of an algorithm. *Usp. Mat. Nauk*. 13:3–28.

Lesk, M. 1975. LEX–a lexical analyzer generator. *Tech. Rep. 39. Bell Labs.* Murray Hill, NJ.

Li, M. and Vitányi, P. 1993. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer–Verlag, Berlin.

McCulloch, W. and Pitts, W. 1943. A logical calculus of ideas immanent in nervous activity. *Bull. Math. Biophys*. 5:115–133.

Post, E. 1943. Formal reductions of the general combinatorial decision problems. *Am. J. Math.* 65:197–215.

Rabin, M. and Scott, D. 1959. Finite automata and their decision problems. *IBM J. Res. Dev*. 3:114–125.

Robinson, R. 1991. Minsky's small universal Turing machine. *Int. J. Math.* 2(5):551–562.

Salomaa, A. 1966. Two complete axiom systems for the algebra of regular events. *J. ACM* 13(1):158–169.

Savitch, J. 1970. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci*. 4(2)177–192.

Schönhage, A. 1980. Storage modification machines. *SIAM J. Comput*. 9:490–508.

Searls, D. 1993. The computational linguistics of biological sequences. In *Artificial Intelligence and Molecular Biology*. L. Hunter, ed., pp. 47–120. MIT Press, Cambridge, MA.

Turing, A. 1936. On computable numbers with an application to the Entscheidungs problem. *Proc. London Math. Soc., Ser. 2* 42:230–265.

van Emde Boas, P. 1990. Machine models and simulations. In *Handbook of Theoretical Computer Science*. J. van Leeuwen, ed., pp. 1–66. Elsevier/MIT Press.

Wood, D. 1987. *Theory of Computation*. Harper and Row.

Further Information

The fundamentals of the theory of computation, automata theory, and formal languages can be found in many text books including Floyd and Beigel [1994], Gurari [1989], Harel [1992], Harrison [1978], Hopcroft and Ullman [1979], and Wood [1987]. The central focus of research in this area is to understand the relationships between the different resource complexity classes. This work is motivated in part by some major open questions about the relationships between resources (such as time and space) and the role of control mechanisms (nondeterminism/randomness). At the same time, new computational models are being introduced and studied. One such recent model that has led to the resolution of a number of interesting problems is the interactive proof systems. They exploit the power of randomness and interaction. Among their applications are new ways to encrypt information as well as some unexpected results about the difficulty of solving some difficult problems even approximately. Another new model is the quantum computational model that incorporates quantum-mechanical effects into the basic move of a Turing machine. There are also attempts to use molecular or cell-level interactions as the basic operations of a computer. Yet another research direction motivated in part by the advances in hardware technology is the study of neural networks, which model (albeit in a simplistic manner) the brain structure of mammals. The following chapters of this volume will present state-of-the-art information about many of these developments. The following annual conferences present the leading research work in computation theory: Association of Computer Machinery (ACM) Annual Symposium on Theory of Computing; Institute of Electrical and Electronics Engineers (IEEE) Symposium on the Foundations of Computer Science; IEEE Conference on Structure in Complexity Theory; International Colloquium on Automata,

Languages and Programming; Symposium on Theoretical Aspects of Computer Science; Mathematical Foundations of Computer Science; and Fundamentals of Computation Theory. There are many related conferences such as Computational Learning Theory, ACM Symposium on Principles of Distributed Computing, etc., where specialized computational models are studied for a specific application area. Concrete algorithms is another closely related area in which the focus is to develop algorithms for specific problems. A number of annual conferences are devoted to this field. We conclude with a list of major journals whose primary focus is in theory of computation: *The Journal of the Association of Computer Machinery, SIAM Journal on Computing, Journal of Computer and System Sciences, Information and Computation, Mathematical Systems Theory, Theoretical Computer Science, Computational Complexity, Journal of Complexity, Information Processing Letters, International Journal of Foundations of Computer Science,* and *ACTA Informatica.*

7

Graph and Network Algorithms

Samir Khuller
University of Maryland

Balaji Raghavachari
University of Texas at Dallas

7.1 Introduction

Graphs are useful in modeling many problems from different scientific disciplines because they capture the basic concept of objects (vertices) and relationships between objects (edges). Indeed, many optimization problems can be formulated in graph theoretic terms. Hence, algorithms on graphs have been widely studied. In this chapter, a few fundamental graph algorithms are described. For a more detailed treatment of graph algorithms, the reader is referred to textbooks on graph algorithms [Cormen et al. 2001, Even 1979, Gibbons 1985, Tarjan 1983].

An undirected *graph* $G = (V, E)$ is defined as a set V of *vertices* and a set E of *edges*. An edge $e = (u, v)$ is an unordered pair of vertices. A *directed graph* is defined similarly, except that its edges are ordered pairs of vertices; that is, for a directed graph, $E \subseteq V \times V$. The terms *nodes* and vertices are used interchangeably. In this chapter, it is assumed that the graph has neither self-loops, edges of the form (v, v), nor multiple edges connecting two given vertices. A graph is a **sparse graph** if $|E| \ll |V|^2$.

Bipartite graphs form a subclass of graphs and are defined as follows. A graph $G = (V, E)$ is bipartite if the vertex set V can be partitioned into two sets X and Y such that $E \subseteq X \times Y$. In other words, each edge of G connects a vertex in X with a vertex in Y. Such a graph is denoted by $G = (X, Y, E)$. Because bipartite graphs occur commonly in practice, algorithms are often specially designed for them.

A vertex w is *adjacent* to another vertex v if $(v, w) \in E$. An edge (v, w) is said to be *incident* on vertices v and w. The *neighbors* of a vertex v are all vertices $w \in V$ such that $(v, w) \in E$. The number of edges incident to a vertex v is called the **degree** of vertex v. For a directed graph, if (v, w) is an edge, then we say that the edge goes from v to w. The *out-degree* of a vertex v is the number of edges from v to other vertices. The *in-degree* of v is the number of edges from other vertices to v.

A **path** $p = [v_0, v_1, \ldots, v_k]$ from v_0 to v_k is a sequence of vertices such that (v_i, v_{i+1}) is an edge in the graph for $0 \leq i < k$. Any edge may be used only once in a path. A **cycle** is a path whose end vertices are the same, that is, $v_0 = v_k$. A path is *simple* if all its internal vertices are distinct. A cycle is *simple* if every node has exactly two edges incident to it in the cycle. A **walk** $w = [v_0, v_1, \ldots, v_k]$ from v_0 to v_k is a sequence of vertices such that (v_i, v_{i+1}) is an edge in the graph for $0 \leq i < k$, in which edges and vertices may be repeated. A walk is *closed* if $v_0 = v_k$. A graph is **connected** if there is a path between every pair of vertices. A directed graph is **strongly connected** if there is a path between every pair of vertices in each direction. An acyclic, undirected graph is a **forest**, and a **tree** is a connected forest. A directed graph without cycles is known as a **directed acyclic graph** (DAG). Consider a binary relation C between the vertices of an undirected graph G such that for any two vertices u and v, uCv if and only if there is a path in G between u and v. It can be shown that C is an equivalence relation, partitioning the vertices of G into equivalence classes, known as the connected components of G.

There are two convenient ways of representing graphs on computers. We first discuss the *adjacency list* representation. Each vertex has a linked list: there is one entry in the list for each of its adjacent vertices. The graph is thus represented as an array of linked lists, one list for each vertex. This representation uses $O(|V| + |E|)$ storage, which is good for sparse graphs. Such a storage scheme allows one to scan all vertices adjacent to a given vertex in time proportional to its degree. The second representation, the *adjacency matrix*, is as follows. In this scheme, an $n \times n$ array is used to represent the graph. The $[i, j]$ entry of this array is 1 if the graph has an edge between vertices i and j, and 0 otherwise. This representation permits one to test if there is an edge between any pair of vertices in constant time. Both these representation schemes can be used in a natural way to represent directed graphs. For all algorithms in this chapter, it is assumed that the given graph is represented by an adjacency list.

Section 7.2 discusses various types of tree traversal algorithms. Sections 7.3 and 7.4 discuss depth-first and breadth-first search techniques. Section 7.5 discusses the single source shortest path problem. Section 7.6 discusses minimum spanning trees. Section 7.7 discusses the bipartite matching problem and the single commodity maximum flow problem. Section 7.8 discusses some traversal problems in graphs, and the Further Information section concludes with some pointers to current research on graph algorithms.

7.2 Tree Traversals

A tree is *rooted* if one of its vertices is designated as the root vertex and all edges of the tree are oriented (directed) to point away from the root. In a rooted tree, there is a directed path from the root to any vertex in the tree. For any directed edge (u, v) in a rooted tree, u is v's *parent* and v is u's *child*. The *descendants* of a vertex w are all vertices in the tree (including w) that are reachable by directed paths starting at w. The *ancestors* of a vertex w are those vertices for which w is a descendant. Vertices that have no children are called **leaves**. A *binary tree* is a special case of a rooted tree in which each node has at most two children, namely, the left child and the right child. The trees rooted at the two children of a node are called the *left subtree* and *right subtree*.

In this section we study techniques for processing the vertices of a given binary tree in various orders. We assume that each vertex of the binary tree is represented by a record that contains fields to hold attributes of that vertex and two special fields *left* and *right* that point to its left and right subtree, respectively.

The three major tree traversal techniques are *preorder*, *inorder*, and *postorder*. These techniques are used as procedures in many tree algorithms where the vertices of the tree have to be processed in a specific order. In a preorder traversal, the root of any subtree has to be processed *before* any of its descendants. In a postorder traversal, the root of any subtree has to be processed *after* all of its descendants. In an inorder traversal, the root of a subtree is processed after all vertices in its left subtree have been processed, but

before any of the vertices in its right subtree are processed. Preorder and postorder traversals generalize to arbitrary rooted trees. In the example to follow, we show how postorder can be used to count the number of descendants of each node and store the value in that node. The algorithm runs in linear time in the size of the tree:

Postorder Algorithm. *PostOrder* (T):

```
if T ≠ nil then
    lc ← PostOrder (left[T]).
    rc ← PostOrder (right[T]).
    desc[T] ← lc + rc + 1.
    return desc[T].
else
    return 0.
end-if
end-proc
```

7.3 Depth-First Search

Depth-first search (DFS) is a fundamental graph searching technique [Tarjan 1972, Hopcroft and Tarjan 1973]. Similar graph searching techniques were given earlier by Tremaux (see Fraenkel [1970] and Lucas [1882]). The structure of DFS enables efficient algorithms for many other graph problems such as biconnectivity, triconnectivity, and planarity [Even 1979].

The algorithm first initializes all vertices of the graph as being unvisited. Processing of the graph starts from an arbitrary vertex, known as the root vertex. Each vertex is processed when it is first discovered (also referred to as *visiting* a vertex). It is first marked as visited, and its adjacency list is then scanned for unvisited vertices. Each time an unvisited vertex is discovered, it is processed recursively by DFS. After a node's entire adjacency list has been explored, that invocation of the DFS procedure returns. This procedure eventually visits all vertices that are in the same connected component of the root vertex. Once DFS terminates, if there are still any unvisited vertices left in the graph, one of them is chosen as the root and the same procedure is repeated.

The set of edges such that each one led to the discovery of a new vertex form a maximal forest of the graph, known as the **DFS forest**; a *maximal forest* of a graph G is an acyclic subgraph of G such that the addition of any other edge of G to the subgraph introduces a cycle. The algorithm keeps track of this forest using parent pointers. In each connected component, only the root vertex has a *nil* parent in the DFS tree.

7.3.1 The Depth-First Search Algorithm

DFS is illustrated using an algorithm that labels vertices with numbers $1, 2, \ldots$ in such a way that vertices in the same component receive the same label. This labeling scheme is a useful preprocessing step in many problems. Each time the algorithm processes a new component, it numbers its vertices with a new label.

Depth-First Search Algorithm. *DFS-Connected-Component* (G):

```
c ← 0.
for all vertices v in G do
    visited[v] ← false.
    finished[v] ← false.
    parent[v] ← nil.
end-for
for all vertices v in G do
    if not visited [v] then
```

```
            c ← c + 1.
            DFS (v, c).
        end-if
    end-for
end-proc

DFS (v, c):
visited[v] ← true.
component[v] ← c.
for all vertices w in adj[v] do
    if not visited[w] then
        parent[w] ← v.
        DFS (w, c).
    end-if
end-for
finished[v] ← true.
end-proc
```

7.3.2 Sample Execution

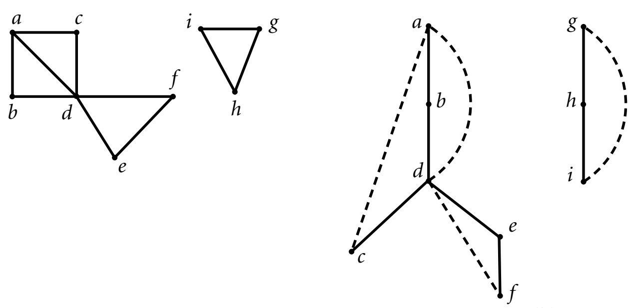
Figure 7.1 shows a graph having two connected components. DFS was started at vertex a, and the DFS forest is shown on the right. DFS visits the vertices b, d, c, e, and f, in that order. DFS then continues with vertices g, h, and i. In each case, the recursive call returns when the vertex has no more unvisited neighbors. Edges (d, a), (c, a), (f, d), and (i, g) are called *back edges* (these do not belong to the DFS forest).

7.3.3 Analysis

A vertex v is processed as soon as it is encountered, and therefore at the start of DFS (v), *visited*$[v]$ is *false*. Since *visited*$[v]$ is set to true as soon as DFS starts execution, each vertex is visited exactly once. Depth-first search processes each edge of the graph exactly twice, once from each of its incident vertices. Since the algorithm spends constant time processing each edge of G, it runs in $O(|V| + |E|)$ time.

Remark 7.1 In the following discussion, there is no loss of generality in assuming that the input graph is connected. For a rooted DFS tree, vertices u and v are said to be *related*, if either u is an ancestor of v, or vice versa.

DFS is useful due to the special way in which the edges of the graph may be classified with respect to a DFS tree. Notice that the DFS tree is not unique, and which edges are added to the tree depends on the



(a) (b)

FIGURE 7.1 Sample execution of DFS on a graph having two connected components: (a) graph, (b) DFS forest.

order in which edges are explored while executing DFS. Edges of the DFS tree are known as *tree* edges. All other edges of the graph are known as *back* edges, and it can be shown that for any edge (u, v), u and v must be related. The graph does not have any *cross* edges, edges that connect two vertices that are unrelated. This property is utilized by a DFS-based algorithm that classifies the edges of a graph into **biconnected** components, maximal subgraphs that cannot be disconnected by the removal of any single vertex [Even 1979].

7.3.4 Directed Depth-First Search

The DFS algorithm extends naturally to directed graphs. Each vertex stores an adjacency list of its outgoing edges. During the processing of a vertex, first mark it as visited, and then scan its adjacency list for unvisited neighbors. Each time an unvisited vertex is discovered, it is processed recursively. Apart from tree edges and back edges (from vertices to their ancestors in the tree), directed graphs may also have *forward* edges (from vertices to their descendants) and *cross* edges (between unrelated vertices). There may be a cross edge (u, v) in the graph only if u is visited after the procedure call DFS (v) has completed execution.

7.3.5 Sample Execution

A sample execution of the directed DFS algorithm is shown in Figure 7.2. DFS was started at vertex a, and the DFS forest is shown on the right. DFS visits vertices b, d, f, and c in that order. DFS then returns and continues with e, and then g. From g, vertices h and i are visited in that order. Observe that (d, a) and (i, g) are back edges. Edges (c, d), (e, d), and (e, f) are cross edges. There is a single forward edge (g, i).

7.3.6 Applications of Depth-First Search

Directed DFS can be used to design a linear-time algorithm that classifies the edges of a given directed graph into *strongly connected* components: maximal subgraphs that have directed paths connecting any pair of vertices in them, in each direction. The algorithm itself involves running DFS twice, once on the original graph, and then a second time on G^R, which is the graph obtained by reversing the direction of all edges in G. During the second DFS, we are able to obtain all of the strongly connected components. The proof of this algorithm is somewhat subtle, and the reader is referred to Cormen et al. [2001] for details.

Checking if a graph has a cycle can be done in linear time using DFS. A graph has a cycle if and only if there exists a back edge relative to any of its depth-first search trees. A directed graph that does not have any cycles is known as a directed acyclic graph. DAGs are useful in modeling precedence constraints in scheduling problems, where nodes denote jobs/tasks, and a directed edge from u to v denotes the constraint that job u must be completed before job v can begin execution. Many problems on DAGs can be solved efficiently using dynamic programming.

A useful concept in DAGs is that of a **topological order**: a linear ordering of the vertices that is consistent with the partial order defined by the edges of the DAG. In other words, the vertices can be labeled with

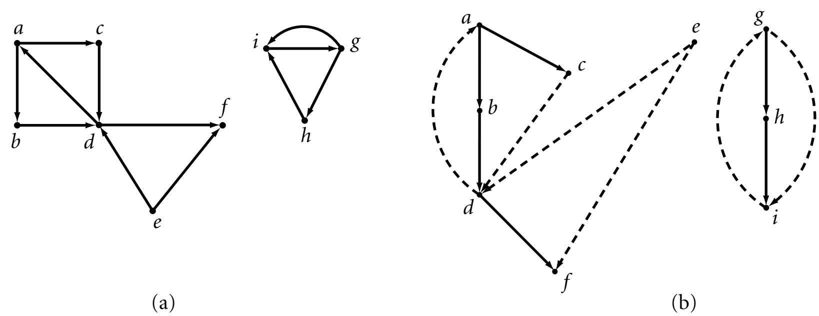


FIGURE 7.2 Sample execution of DFS on a directed graph: (a) graph, (b) DFS forest.

distinct integers in the range $[1 \ldots |V|]$ such that if there is a directed edge from a vertex labeled i to a vertex labeled j, then $i < j$. The vertices of a given DAG can be ordered topologically in linear time by a suitable modification of the DFS algorithm. We keep a counter whose initial value is $|V|$. As each vertex is marked finished, we assign the counter value as its topological number and decrement the counter. Observe that there will be no back edges; and that for all edges (u, v), v will be marked finished before u. Thus, the topological number of v will be higher than that of u. Topological sort has applications in diverse areas such as project management, scheduling, and circuit evaluation.

7.4 Breadth-First Search

Breadth-first search (BFS) is another natural way of searching a graph. The search starts at a root vertex r. Vertices are added to a queue as they are discovered, and processed in (first-in–first-out) (FIFO) order.

Initially, all vertices are marked as unvisited, and the queue consists of only the root vertex. The algorithm repeatedly removes the vertex at the front of the queue, and scans its neighbors in the graph. Any neighbor not visited is added to the end of the queue. This process is repeated until the queue is empty. All vertices in the same connected component as the root are scanned and the algorithm outputs a spanning tree of this component. This tree, known as a **breadth-first tree**, is made up of the edges that led to the discovery of new vertices. The algorithm labels each vertex v by $d[v]$, the distance (length of a shortest path) of v from the root vertex, and stores the BFS tree in the array p, using parent pointers. Vertices can be partitioned into levels based on their distance from the root. Observe that edges not in the BFS tree always go either between vertices in the same level, or between vertices in adjacent levels. This property is often useful.

Breadth-First Search Algorithm. *BFS-Distance* (G, r):

```
MakeEmptyQueue (Q).
for all vertices v in G do
   visited[v] ← false.
   d[v] ← ∞.
   p[v] ← nil.
end-for
visited[r] ← true.
d[r] ← 0.
Enqueue (Q, r).
while not Empty (Q) do
   v ← Dequeue (Q).
   for all vertices w in adj[v] do
      if not visited [w] then
         visited[w] ← true.
         p[w] ← v.
         d[w] ← d[v] + 1.
         Enqueue (Q, w).
      end-if
   end-for
end-while
end-proc
```

7.4.1 Sample Execution

Figure 7.3 shows a connected graph on which BFS was run with vertex a as the root. When a is processed, vertices b, d, and c are added to the queue. When b is processed, nothing is done since all its neighbors have been visited. When d is processed, e and f are added to the queue. Finally c, e, and f are processed.

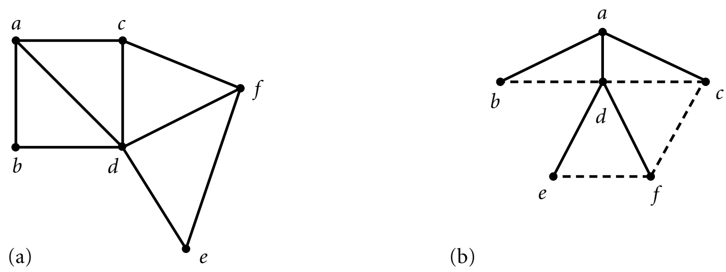


FIGURE 7.3 Sample execution of BFS on a graph: (a) graph, (b) BFS tree.

7.4.2 Analysis

There is no loss of generality in assuming that the graph G is connected, since the algorithm can be repeated in each connected component, similar to the DFS algorithm. The algorithm processes each vertex exactly once, and each edge exactly twice. It spends a constant amount of time in processing each edge. Hence, the algorithm runs in $O(|V| + |E|)$ time.

7.5 Single-Source Shortest Paths

A natural problem that often arises in practice is to compute the shortest paths from a specified node to all other nodes in an undirected graph. BFS solves this problem if all edges in the graph have the same length. Consider the more general case when each edge is given an arbitrary, non-negative length, and one needs to calculate a shortest length path from the root vertex to all other nodes of the graph, where the length of a path is defined to be the sum of the lengths of its edges. The distance between two nodes is the length of a shortest path between them.

7.5.1 Dijkstra's Algorithm

Dijkstra's algorithm [Dijkstra 1959] provides an efficient solution to this problem. For each vertex v, the algorithm maintains an upper bound to the distance from the root to vertex v in $d[v]$; initially $d[v]$ is set to infinity for all vertices except the root. The algorithm maintains a set S of vertices with the property that for each vertex $v \in S$, $d[v]$ is the length of a shortest path from the root to v. For each vertex u in $V - S$, the algorithm maintains $d[u]$, the shortest known distance from the root to u that goes entirely within S, except for the last edge. It selects a vertex u in $V - S$ of minimum $d[u]$, adds it to S, and updates the distance estimates to the other vertices in $V - S$. In this update step, it checks to see if there is a shorter path to any vertex in $V - S$ from the root that goes through u. Only the distance estimates of vertices that are adjacent to u are updated in this step. Because the primary operation is the selection of a vertex with minimum distance estimate, a priority queue is used to maintain the d-values of vertices. The priority queue should be able to handle a DecreaseKey operation to update the d-value in each iteration. The next algorithm implements Dijkstra's algorithm.

Dijkstra's Algorithm. *Dijkstra-Shortest Paths* (G, r):

```
for all vertices v in G do
   visited[v] ← false.
   d[v] ← ∞.
   p[v] ← nil.
end-for
d[r] ← 0.
BuildPQ (H, d).
while not Empty (H) do
```

```
 9    u ← DeleteMin (H).
10    visited[u] ← true.
11    for all vertices v in adj[u] do
12      Relax (u, v).
13    end-for
14  end-while
end-proc
```

```
Relax (u, v)
1 if not visited[v] and d[v] > d[u] + w(u, v) then
2   d[v] ← d[u] + w(u, v).
3   p[v] ← u.
4   DecreaseKey (H, v, d[v]).
5 end-if
end-proc
```

7.5.1.1 Sample Execution

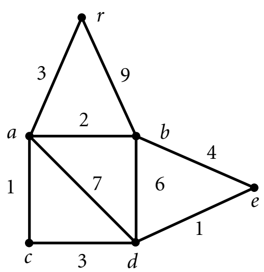
Figure 7.4 shows a sample execution of the algorithm. The column titled Iter specifies the number of iterations that the algorithm has executed through the **while** loop in step 8. In iteration 0, the initial values of the distance estimates are ∞. In each subsequent line of the table, the column marked u shows the vertex that was chosen in step 9 of the algorithm, and the change to the distance estimates at the end of that iteration of the **while** loop. In the first iteration, vertex r was chosen, after that a was chosen because it had the minimum distance label among the unvisited vertices, and so on. The distance labels of the unvisited neighbors of the visited vertex are updated in each iteration.

7.5.1.2 Analysis

The running time of the algorithm depends on the data structure that is used to implement the priority queue H. The algorithm performs $|V|$ DELETEMIN operations and, at most, $|E|$ DECREASEKEY operations. If a binary heap is used to update the records of any given vertex, each of these operations runs in $O(\log |V|)$ time. There is no loss of generality in assuming that the graph is connected. Hence, the algorithm runs in $O(|E| \log |V|)$. If a Fibonacci heap is used to implement the priority queue, the running time of the algorithm is $O(|E| + |V| \log |V|)$. Although the Fibonacci heap gives the best asymptotic running time, the binary heap implementation is likely to give better running times for most practical instances.

7.5.2 Bellman–Ford Algorithm

The shortest path algorithm described earlier directly generalizes to directed graphs, but it does not work correctly if the graph has edges of negative length. For graphs that have edges of negative length, but no



Iter	u	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$
0	—	∞	∞	∞	∞	∞
1	r	3	9	∞	∞	∞
2	a	3	5	4	10	∞
3	c	3	5	4	7	∞
4	b	3	5	4	7	9
5	d	3	5	4	7	8
6	e	3	5	4	7	8

FIGURE 7.4 Dijkstra's shortest path algorithm.

cycles of negative length, there is a different algorithm due to Bellman [1958] and Ford and Fulkerson [1962] that solves the single source shortest paths problem in $O(|V||E|)$ time.

The key to understanding this algorithm is the RELAX operation applied to an edge. In a single scan of the edges, we execute the RELAX operation on each edge. We then repeat the step $|V| - 1$ times. No special data structures are required to implement this algorithm, and the proof relies on the fact that a shortest path is simple and contains at most $|V| - 1$ edges (see Cormen et al. [2001] for a proof).

This problem also finds applications in finding a feasible solution to a system of linear equations, where each equation specifies a bound on the difference of two variables. Each constraint is modeled by an edge in a suitably defined directed graph. Such systems of equations arise in real-time applications.

7.6 Minimum Spanning Trees

The following fundamental problem arises in network design. A set of sites needs to be connected by a network. This problem has a natural formulation in graph-theoretic terms. Each site is represented by a vertex. Edges between vertices represent a potential link connecting the corresponding nodes. Each edge is given a nonnegative cost corresponding to the cost of constructing that link. A tree is a minimal network that connects a set of nodes. The cost of a tree is the sum of the costs of its edges. A minimum-cost tree connecting the nodes of a given graph is called a minimum-cost spanning tree, or simply a **minimum spanning tree**.

The problem of computing a minimum spanning tree (MST) arises in many areas, and as a subproblem in combinatorial and geometric problems. MSTs can be computed efficiently using algorithms that are greedy in nature, and there are several different algorithms for finding an MST. One of the first algorithms was due to Boruvka [1926]. The two algorithms that are popularly known as Prim's algorithm and Kruskal's algorithm are described here. (Prim's algorithm was first discovered by Jarnik [1930].)

7.6.1 Prim's Algorithm

Prim's [1957] algorithm for finding an MST of a given graph is one of the oldest algorithms to solve the problem. The basic idea is to start from a single vertex and gradually grow a tree, which eventually spans the entire graph. At each step, the algorithm has a tree that covers a set S of vertices, and looks for a *good* edge that may be used to extend the tree to include a vertex that is currently not in the tree. All edges that go from a vertex in S to a vertex in $V - S$ are candidate edges. The algorithm selects a minimum-cost edge from these candidate edges and adds it to the current tree, thereby adding another vertex to S.

As in the case of Dijkstra's algorithm, each vertex $u \in V - S$ can attach itself to only one vertex in the tree (so that cycles are not generated in the solution). Because the algorithm always chooses a minimum-cost edge, it needs to maintain a minimum-cost edge that connects u to some vertex in S as the candidate edge for including u in the tree. A priority queue of vertices is used to select a vertex in $V - S$ that is incident to a minimum-cost candidate edge.

Prim's Algorithm. *Prim-MST* (G, r):

```
for all vertices v in G do
   visited[v] ← false.
   d[v] ← ∞.
   p[v] ← nil.
end-for
d[r] ← 0.
BuildPQ (H, d).
while not Empty (H) do
   u ← DeleteMin (H).
   visited[u] ← true.
   for all vertices v in adj[u] do
      if not visited[v] and d[v] > w(u,v) then
```

```
        d[v] ← w(u,v).
        p[v] ← u.
        DecreaseKey (H, v, d[v]).
      end-if
    end-for
  end-while
end-proc
```

7.6.1.1 Analysis

First observe the similarity between Prim's and Dijkstra's algorithms. Both algorithms start building the tree from a single vertex and grow it by adding one vertex at a time. The only difference is the rule for deciding when the current label is updated for vertices outside the tree. Both algorithms have the same structure and therefore have similar running times. Prim's algorithm runs in $O(|E| \log |V|)$ time if the priority queue is implemented using binary heaps, and it runs in $O(|E| + |V| \log |V|)$ if the priority queue is implemented using Fibonacci heaps.

7.6.2 Kruskal's Algorithm

Kruskal's [1956] algorithm for finding an MST of a given graph is another classical algorithm for the problem, and is also greedy in nature. Unlike Prim's algorithm, which grows a single tree, Kruskal's algorithm grows a forest. First, the edges of the graph are sorted in nondecreasing order of their costs. The algorithm starts with the empty spanning forest (no edges). The edges of the graph are scanned in sorted order, and if the addition of the current edge does not generate a cycle in the current forest, it is added to the forest. The main test at each step is: does the current edge connect two vertices in the same connected component? Eventually, the algorithm adds $|V| - 1$ edges to make a spanning tree in the graph.

The main data structure needed to implement the algorithm is for the maintenance of connected components, to ensure that the algorithm does not add an edge between two nodes in the same connected component. An abstract version of this problem is known as the Union-Find problem for a collection of disjoint sets. Efficient algorithms are known for this problem, where an arbitrary sequence of UNION and FIND operations can be implemented to run in almost linear time [Cormen et al. 2001, Tarjan 1983].

Kruskal's Algorithm. *Kruskal-MST* (G):

```
1 T ← φ.
2 for all vertices v in G do
3   Makeset(v).
4 Sort the edges of G by nondecreasing order of costs.
5 for all edges e = (u,v) in G in sorted order do
6   if Find (u) ≠ Find (v) then
7     T ← T ∪ (u,v).
8     Union (u, v).
9 end-proc
```

7.6.2.1 Analysis

The running time of the algorithm is dominated by step 4 of the algorithm in which the edges of the graph are sorted by nondecreasing order of their costs. This takes $O(|E| \log |E|)$ [which is also $O(|E| \log |V|)$] time using an efficient sorting algorithm such as Heap-sort. Kruskal's algorithm runs faster in the following special cases: if the edges are presorted, if the edge costs are within a small range, or if the number of different edge costs is bounded by a constant. In all of these cases, the edges can be sorted in linear time, and the algorithm runs in near-linear time, $O(|E| \alpha(|E|, |V|))$, where $\alpha(m, n)$ is the inverse Ackermann function [Tarjan 1983].

Remark 7.2 The MST problem can be generalized to directed graphs. The equivalent of trees in directed graphs are called **arborescences** or **branchings**; and because edges have directions, they are rooted spanning trees. An incoming branching has the property that every vertex has a unique path to the root. An outgoing branching has the property that there is a unique path from the root to each vertex in the graph. The input is a directed graph with arbitrary costs on the edges and a root vertex r. The output is a minimum-cost branching rooted at r. The algorithms discussed in this section for finding minimum spanning trees do not directly extend to the problem of finding optimal branchings. There are efficient algorithms that run in $O(|E|+|V|\log|V|)$ time using Fibonacci heaps for finding minimum-cost branchings [Gibbons 1985, Gabow et al. 1986]. These algorithms are based on techniques for weighted matroid intersection [Lawler 1976]. Almost linear-time deterministic algorithms for the MST problem in undirected graphs are also known [Fredman and Tarjan 1987].

7.7 Matchings and Network Flows

Networks are important both for electronic communication and for transporting goods. The problem of efficiently moving entities (such as bits, people, or products) from one place to another in an underlying network is modeled by the **network flow** problem. The problem plays a central role in the fields of operations research and computer science, and much emphasis has been placed on the design of efficient algorithms for solving it. Many of the basic algorithms studied earlier in this chapter play an important role in developing various implementations for network flow algorithms.

First the **matching** problem, which is a special case of the flow problem, is introduced. Then the **assignment problem**, which is a generalization of the matching problem to the weighted case, is studied. Finally, the network flow problem is introduced and algorithms for solving it are outlined.

The maximum matching problem is studied here in detail only for bipartite graphs. Although this restricts the class of graphs, the same principles are used to design polynomial time algorithms for graphs that are not necessarily bipartite. The algorithms for general graphs are complex due to the presence of structures called *blossoms*, and the reader is referred to Papadimitriou and Steiglitz [1982, Chapter 10], or Tarjan [1983, Chapter 9] for a detailed treatment of how blossoms are handled. Edmonds (see Even [1979]) gave the first algorithm to solve the matching problem in polynomial time. Micali and Vazirani [1980] obtained an $O(\sqrt{|V|}|E|)$ algorithm for nonbipartite matching by extending the algorithm by Hopcroft and Karp [1973] for the bipartite case.

7.7.1 Matching Problem Definitions

Given a graph $G = (V, E)$, a matching M is a subset of the edges such that no two edges in M share a common vertex. In other words, the problem is that of finding a set of independent edges that have no incident vertices in common. The cardinality of M is usually referred to as its *size*.

The following terms are defined with respect to a matching M. The edges in M are called *matched edges* and edges not in M are called *free edges*. Likewise, a vertex is a *matched vertex* if it is incident to a matched edge. A *free vertex* is one that is not matched. The *mate* of a matched vertex v is its neighbor w that is at the other end of the matched edge incident to v. A matching is called *perfect* if all vertices of the graph are matched in it. The objective of the maximum matching problem is to maximize $|M|$, the size of the matching. If the edges of the graph have weights, then the *weight* of a matching is defined to be the sum of the weights of the edges in the matching. A path $p = [v_1, v_2, \ldots, v_k]$ is called an *alternating path* if the edges (v_{2j-1}, v_{2j}), $j = 1, 2, \ldots$, are free and the edges (v_{2j}, v_{2j+1}), $j = 1, 2, \ldots$, are matched. An **augmenting path** $p = [v_1, v_2, \ldots, v_k]$ is an alternating path in which both v_1 and v_k are free vertices. Observe that an augmenting path is defined with respect to a specific matching. The symmetric difference of a matching M and an augmenting path P, $M \oplus P$, is defined to be $(M - P) \cup (P - M)$. The operation can be generalized to the case when P is any subset of the edges.

7.7.2 Applications of Matching

Matchings are the underlying basis for many optimization problems. Problems of assigning workers to jobs can be naturally modeled as a bipartite matching problem. Other applications include assigning a collection of jobs with precedence constraints to two processors, such that the total execution time is minimized [Lawler 1976]. Other applications arise in chemistry, in determining structure of chemical bonds, matching moving objects based on a sequence of photographs, and localization of objects in space after obtaining information from multiple sensors [Ahuja et al. 1993].

7.7.3 Matchings and Augmenting Paths

The following theorem gives necessary and sufficient conditions for the existence of a perfect matching in a bipartite graph.

Theorem 7.1 (Hall's Theorem.) *A bipartite graph $G = (X, Y, E)$ with $|X| = |Y|$ has a perfect matching if and only if $\forall S \subseteq X, |N(S)| \geq |S|$, where $N(S) \subseteq Y$ is the set of vertices that are neighbors of some vertex in S.*

Although Theorem 7.1 captures exactly the conditions under which a given bipartite graph has a perfect matching, it does not lead directly to an algorithm for finding maximum matchings. The following lemma shows how an augmenting path with respect to a given matching can be used to increase the size of a matching. An efficient algorithm that uses augmenting paths to construct a maximum matching incrementally is described later.

Lemma 7.1 *Let P be the edges on an augmenting path $p = [v_1, \ldots, v_k]$ with respect to a matching M. Then $M' = M \oplus P$ is a matching of cardinality $|M| + 1$.*

***Proof* 7.1** Since P is an augmenting path, both v_1 and v_k are free vertices in M. The number of free edges in P is one more than the number of matched edges. The symmetric difference operator replaces the matched edges of M in P by the free edges in P. Hence, the size of the resulting matching, $|M'|$, is one more than $|M|$. □

The following theorem provides a necessary and sufficient condition for a given matching M to be a maximum matching.

Theorem 7.2 *A matching M in a graph G is a maximum matching if and only if there is no augmenting path in G with respect to M.*

***Proof* 7.2** If there is an augmenting path with respect to M, then M cannot be a maximum matching, since by Lemma 7.1 there is a matching whose size is larger than that of M. To prove the converse we show that if there is no augmenting path with respect to M, then M is a maximum matching. Suppose that there is a matching M' such that $|M'| > |M|$. Consider the set of edges $M \oplus M'$. These edges form a subgraph in G. Each vertex in this subgraph has degree at most two, since each node has at most one edge from each matching incident to it. Hence, each connected component of this subgraph is either a path or a simple cycle. For each cycle, the number of edges of M is the same as the number of edges of M'. Since $|M'| > |M|$, one of the paths must have more edges from M' than from M. This path is an augmenting path in G with respect to the matching M, contradicting the assumption that there were no augmenting paths with respect to M. □

7.7.4 Bipartite Matching Algorithm

7.7.4.1 High-Level Description

The algorithm starts with the empty matching $M = \emptyset$, and augments the matching in phases. In each phase, an augmenting path with respect to the current matching M is found, and it is used to increase the size of the matching. An augmenting path, if one exists, can be found in $O(|E|)$ time, using a procedure similar to breadth-first search described in Section 7.4.

The search for an augmenting path proceeds from the free vertices. At each step when a vertex in X is processed, all its unvisited neighbors are also searched. When a matched vertex in Y is considered, only its matched neighbor is searched. This search proceeds along a subgraph referred to as the *Hungarian tree*.

Initially, all free vertices in X are placed in a queue that holds vertices that are yet to be processed. The vertices are removed one by one from the queue and processed as follows. In turn, when vertex v is removed from the queue, the edges incident to it are scanned. If it has a neighbor in the vertex set Y that is free, then the search for an augmenting path is successful; procedure AUGMENT is called to update the matching, and the algorithm proceeds to its next phase. Otherwise, add the mates of all of the matched neighbors of v to the queue if they have never been added to the queue, and continue the search for an augmenting path. If the algorithm empties the queue without finding an augmenting path, its current matching is a maximum matching and it terminates.

The main data structure that the algorithm uses consists of the arrays *mate* and *free*. The array *mate* is used to represent the current matching. For a matched vertex $v \in G$, *mate*$[v]$ denotes the matched neighbor of vertex v. For $v \in X$, *free*$[v]$ is a vertex in Y that is adjacent to v and is free. If no such vertex exists, then *free*$[v] = 0$.

Bipartite Matching Algorithm. *Bipartite Matching* $(G = (X, Y, E))$:

```
for all vertices v in G do
   mate[v] ← 0.
end-for
found ← false.
while not found do
     Initialize.
     MakeEmptyQueue (Q).
     for all vertices x ∈ X do
        if mate[x] = 0 then
           Enqueue (Q,x).
           label[x] ← 0.
        endif
     end-for
     done ← false.
     while not done and not Empty (Q) do
        x ← Dequeue (Q).
        if free[x] ≠ 0 then
           Augment(x).
           done ← true.
        else
           for all edges (x,x') ∈ A do
              if label[x'] = 0 then
                 label[x'] ← x.
                 Enqueue (Q,x').
              end-if
           end-for
```

```
            end-if
            if Empty (Q) then
               found ← true.
            end-if
         end-while
   end-while
end-proc
```

Initialize :

```
for all vertices x ∈ X do
   free[x] ← 0.
end-for
A ← ∅.
for all edges (x,y) ∈ E do
   if mate[y] = 0 then free[x] ← y
   else if mate[y] ≠ x then A ← A ∪ (x, mate[y]).
   end-if
end-for
end-proc
```

Augment(x):

```
if label[x] = 0 then
   mate[x] ← free[x].
   mate[free[x]] ← x
else
   free[label[x]] ← mate[x]
   mate[x] ← free[x]
   mate[free[x]] ← x
   Augment (label[x])
end-if
end-proc
```

7.7.4.2 Sample Execution

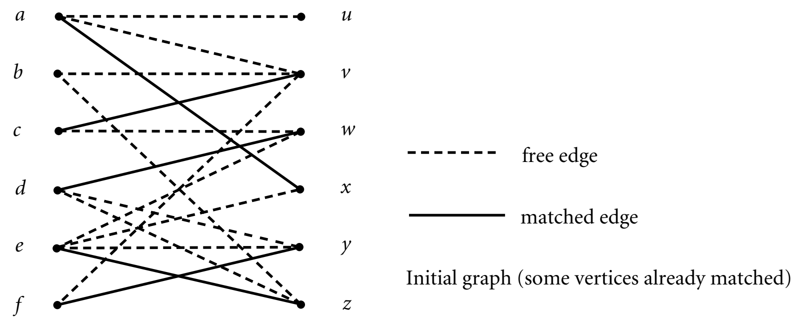
Figure 7.5 shows a sample execution of the matching algorithm. We start with a partial matching and show the structure of the resulting Hungarian tree. An augmenting path from vertex b to vertex u is found by the algorithm.

7.7.4.3 Analysis

If there are augmenting paths with respect to the current matching, the algorithm will find at least one of them. Hence, when the algorithm terminates, the graph has no augmenting paths with respect to the current matching and the current matching is optimal. Each iteration of the main **while** loop of the algorithm runs in $O(|E|)$ time. The construction of the auxiliary graph A and computation of the array *free* also take $O(|E|)$ time. In each iteration, the size of the matching increases by one and thus there are, at most, $\min(|X|, |Y|)$ iterations of the **while** loop. Therefore, the algorithm solves the matching problem for bipartite graphs in time $O(\min(|X|, |Y|)|E|)$. Hopcroft and Karp [1973] showed how to improve the running time by finding a maximal set of shortest disjoint augmenting paths in a single phase in $O(|E|)$ time. They also proved that the algorithm runs in only $O(\sqrt{|V|})$ phases.

7.7.5 Assignment Problem

We now introduce the assignment problem, which is that of finding a maximum-weight matching in a given bipartite graph in which edges are given nonnegative weights. There is no loss of generality in



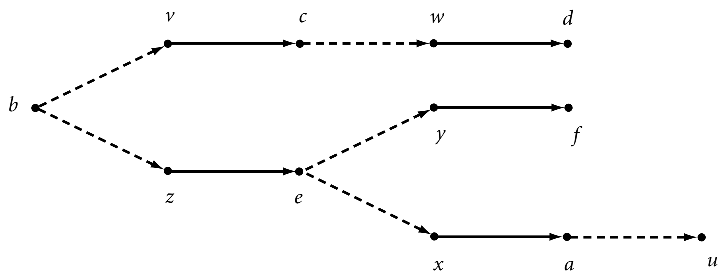


FIGURE 7.5 Sample execution of matching algorithm.

assuming that the graph is complete, since zero-weight edges may be added between pairs of vertices that are nonadjacent in the original graph without affecting the weight of a maximum-weight matching. The minimum-weight perfect matching can be reduced to the maximum-weight matching problem as follows: choose a constant M that is larger than the weight of any edge. Assign each edge a new weight of $w'(e) = M - w(e)$. Observe that maximum-weight matchings with the new weight function are minimum-weight perfect matchings with the original weights. We restrict our attention to the study of the maximum-weight matching problem for bipartite graphs. Similar techniques have been used to solve the maximum-weight matching problem in arbitrary graphs (see Lawler [1976] and Papadimitriou and Steiglitz [1982]).

The input is a complete bipartite graph $G = (X, Y, X \times Y)$ and each edge e has a nonnegative weight of $w(e)$. The following algorithm, known as the Hungarian method, was first given by Kuhn [1955]. The method can be viewed as a primal-dual algorithm in the linear programming framework [Papadimitriou and Steiglitz 1982]. No knowledge of linear programming is assumed here.

A *feasible vertex-labeling* ℓ is defined to be a mapping from the set of vertices in G to the real numbers such that for each edge (x_i, y_j) the following condition holds:

$$\ell(x_i) + \ell(y_j) \geq w(x_i, y_j)$$

The following can be verified to be a feasible vertex labeling. For each vertex $y_j \in Y$, set $\ell(y_j)$ to be 0; and for each vertex $x_i \in X$, set $\ell(x_i)$ to be the maximum weight of an edge incident to x_i,

$$\ell(y_j) = 0,$$
$$\ell(x_i) = \max_j w(x_i, y_j)$$

The *equality subgraph*, G_ℓ, is defined to be the subgraph of G, which includes all vertices of G but only those edges (x_i, y_j) that have weights such that

$$\ell(x_i) + \ell(y_j) = w(x_i, y_j)$$

The connection between equality subgraphs and maximum-weighted matchings is established by the following theorem.

Theorem 7.3 *If the equality subgraph, G_ℓ, has a perfect matching, M^*, then M^* is a maximum-weight matching in G.*

***Proof* 7.3** Let M^* be a perfect matching in G_ℓ. By definition,

$$w(M^*) = \sum_{e \in M^*} w(e) = \sum_{v \in X \cup Y} \ell(v)$$

Let M be any perfect matching in G. Then,

$$w(M) = \sum_{e \in M} w(e) \leq \sum_{v \in X \cup Y} \ell(v) = w(M^*)$$

Hence, M^* is a maximum-weight perfect matching. □

7.7.5.1 High-Level Description

Theorem 7.3 is the basis of the algorithm for finding a maximum-weight matching in a complete bipartite graph. The algorithm starts with a feasible labeling, then computes the equality subgraph and a maximum cardinality matching in this subgraph. If the matching found is perfect, by Theorem 7.3 the matching must be a maximum-weight matching and the algorithm returns it as its output. Otherwise, more edges need to be added to the equality subgraph by *revising* the vertex labels. The revision keeps edges from the current matching in the equality subgraph. After more edges are added to the equality subgraph, the algorithm grows the Hungarian trees further. Either the size of the matching increases because an augmenting path is found, or a new vertex is added to the Hungarian tree. In the former case, the current phase terminates and the algorithm starts a new phase, because the matching size has increased. In the latter case, new nodes are added to the Hungarian tree. In n phases, the tree includes all of the nodes, and therefore there are at most n phases before the size of the matching increases.

It is now described in more detail how the labels are updated and which edges are added to the equality subgraph G_ℓ. Suppose M is a maximum matching in G_ℓ found by the algorithm. Hungarian trees are grown from all the free vertices in X. Vertices of X (including the free vertices) that are encountered in the search are added to a set S, and vertices of Y that are encountered in the search are added to a set T. Let $\overline{S} = X - S$ and $\overline{T} = Y - T$. Figure 7.6 illustrates the structure of the sets S and T. Matched edges are shown in bold; the other edges are the edges in G_ℓ. Observe that there are no edges in the equality subgraph from S to $\overline{T}$, although there may be edges from T to $\overline{S}$. Let us choose δ to be the smallest value such that some edge of $G - G_\ell$ enters the equality subgraph. The algorithm now revises the labels as follows. Decrease all of the labels of vertices in S by δ and increase the labels of the vertices in T by δ. This ensures that edges in the matching continue to stay in the equality subgraph. Edges in G (not in G_ℓ) that go from vertices in S to vertices in $\overline{T}$ are candidate edges to enter the equality subgraph, since one label is decreasing and the other is unchanged. Suppose this edge goes from $x \in S$ to $y \in \overline{T}$. If y is free, then an augmenting path has been found. On the other hand, if y is matched, the Hungarian tree is grown by moving y to T and its matched neighbor to S, and the process of revising labels continues.

7.7.6 B-Matching Problem

The B-Matching problem is a generalization of the matching problem. In its simplest form, given an integer $b \geq 1$, the problem is to find a subgraph H of a given graph G such that the degree of each vertex is exactly equal to b in H (such a subgraph is called a *b-regular subgraph*). The problem can also be formulated as an optimization problem by seeking a subgraph H with most edges, with the degree of each vertex to

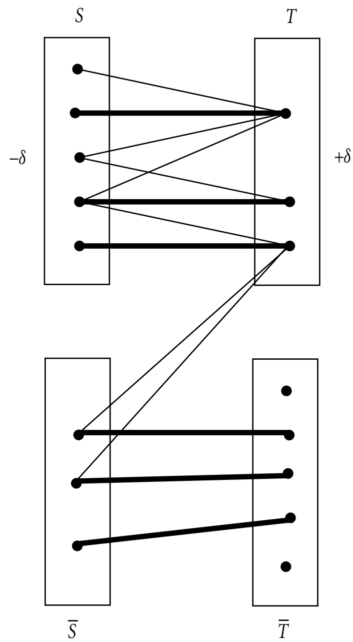


FIGURE 7.6 Sets S and T as maintained by the algorithm. Only edges in G_ℓ are shown.

be at most b in H. Several generalizations are possible, including different degree bounds at each vertex, degrees of some vertices unspecified, and edges with weights. All variations of the B-Matching problem can be solved using the techniques for solving the Matching problem.

In this section, we show how the problem can be solved for the unweighted B-Matching problem in which each vertex v is given a degree bound of $b[v]$, and the objective is to find a subgraph H in which the degree of each vertex v is exactly equal to $b[v]$. From the given graph G, construct a new graph G_b as follows. For each vertex $v \in G$, introduce $b[v]$ vertices in G_b, namely $v_1, v_2, \ldots, v_{b[v]}$. For each edge $e = (u, v)$ in G, add two new vertices e_u and e_v to G_b, along with the edge (e_u, e_v). In addition, add edges between v_i and e_v, for $1 \le i \le b[v]$ (and between u_j and e_u, for $1 \le j \le b[u]$). We now show that there is a natural one-to-one correspondence between B-Matchings in G and perfect matchings in G_b.

Given a B-Matching H in G, we show how to construct a perfect matching in G_b. For each edge $(u, v) \in H$, match e_u to the next available u_j, and e_v to the next available v_i. Since u is incident to exactly $b[u]$ edges in H, there are exactly enough nodes $u_1, u_2 \ldots u_{b[v]}$ in the previous step. For all edges $e = (u, v) \in G - H$, we match e_u and e_v. It can be verified that this yields a perfect matching in G_b.

We now show how to construct a B-Matching in G, given a perfect matching in G_b. Let M be a perfect matching in G_b. For each edge $e = (u, v) \in G$, if $(e_u, e_b) \in M$, then do not include the edge e in the B-Matching. Otherwise, e_u is matched to some u_j and e_v is matched to some v_i in M. In this case, we include e in our B-Matching. Since there are exactly $b[u]$ vertices $u_1, u_2, \ldots u_{b[u]}$, each such vertex introduces an edge into the B-Matching, and therefore the degree of u is exactly $b[u]$. Therefore, we get a B-Matching in G.

7.7.7 Network Flows

A number of polynomial time flow algorithms have been developed over the past two decades. The reader is referred to Ahuja et al. [1993] for a detailed account of the historical development of the various flow methods. Cormen et al. [2001] review the preflow push method in detail; and to complement their coverage, an implementation of the blocking flow technique of Malhotra et al. [1978] is discussed here.

7.7.8 Network Flow Problem Definitions

First the network flow problem and its basic terminology are defined.

Flow network: A flow network $G = (V, E)$ is a directed graph, with two specially marked nodes, namely, the source s and the sink t. There is a *capacity* function $c : E \mapsto R^+$ that maps edges to positive real numbers.

Max-flow problem: A flow function $f : E \mapsto R$ maps edges to real numbers. For an edge $e = (u, v)$, $f(e)$ refers to the flow on edge e, which is also called the net flow from vertex u to vertex v. This notation is extended to sets of vertices as follows: If X and Y are sets of vertices then $f(X, Y)$ is defined to be $\sum_{x\in X}\sum_{y\in Y} f(x, y)$. A flow function is required to satisfy the following constraints:

- *Capacity constraint.* For all edges e, $f(e) \leq c(e)$.
- *Skew symmetry constraint.* For an edge $e = (u, v)$, $f(u, v) = -f(v, u)$.
- *Flow conservation.* For all vertices $u \in V - \{s, t\}$, $\sum_{v\in V} f(u, v) = 0$.

The capacity constraint says that the total flow on an edge does not exceed its capacity. The skew symmetry condition says that the flow on an edge is the negative of the flow in the reverse direction. The flow conservation constraint says that the total net flow out of any vertex other than the source and sink is zero.

The *value* of the flow is defined as

$$|f| = \sum_{v\in V} f(s, v)$$

In other words, it is the net flow out of the source. In the *maximum-flow problem*, the objective is to find a flow function that satisfies the three constraints, and also maximizes the total flow value $|f|$.

Remark 7.3 This formulation of the network flow problem is powerful enough to capture generalizations where there are many sources and sinks (single commodity flow), and where both vertices and edges have capacity constraints, etc.

First, the notion of cuts is defined, and the max-flow min-cut theorem is introduced. Then, residual networks, layered networks, and the concept of blocking flows are introduced. Finally, an efficient algorithm for finding a blocking flow is described.

An ***s–t* cut** of the graph is a partitioning of the vertex set V into two sets S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the net flow across the cut is defined as $f(S, T)$. The capacity of the cut is similarly defined as $c(S, T) = \sum_{x\in X}\sum_{y\in Y} c(x, y)$. The net flow across a cut may include negative net flows between vertices, but the capacity of the cut includes only nonnegative values, that is, only the capacities of edges from S to T.

Using the flow conservation principle, it can be shown that the net flow across an s–t cut is exactly the flow value $|f|$. By the capacity constraint, the flow across the cut cannot exceed the capacity of the cut. Thus, the value of the maximum flow is no greater than the capacity of a minimum s–t cut. The well-known *max-flow min-cut theorem* [Elias et al. 1956, Ford and Fulkerson 1962] proves that the two numbers are actually equal. In other words, if f^* is a maximum flow, then there is some cut $(X, \overline{X})$ such that $|f^*| = c(X, \overline{X})$. The reader is referred to Cormen et al. [2001] and Tarjan [1983] for further details.

The *residual capacity* of a flow f is defined to be a function on vertex pairs given by $c'(v, w) = c(v, w) - f(v, w)$. The residual capacity of an edge (v, w), $c'(v, w)$, is the number of additional units of flow that can be pushed from v to w without violating the capacity constraints. An edge e is *saturated* if $c(e) = f(e)$, that is, if its residual capacity, $c'(e)$, is zero. The residual graph $G_R(f)$ for a flow f is the graph with vertex set V, source and sink s and t, respectively, and those edges (v, w) for which $c'(v, w) > 0$.

An augmenting path for f is a path P from s to t in $G_R(f)$. The residual capacity of P, denoted by $c'(P)$, is the minimum value of $c'(v, w)$ over all edges (v, w) in the path P. The flow can be increased by $c'(P)$, by increasing the flow on each edge of P by this amount. Whenever $f(v, w)$ is changed, $f(w, v)$ is also correspondingly changed to maintain skew symmetry.

Most flow algorithms are based on the concept of augmenting paths pioneered by Ford and Fulkerson [1956]. They start with an initial zero flow and augment the flow in stages. In each stage, a residual graph $G_R(f)$ with respect to the current flow function f is constructed and an augmenting path in $G_R(f)$ is found to increase the value of the flow. Flow is increased along this path until an edge in this path is saturated. The algorithms iteratively keep increasing the flow until there are no more augmenting paths in $G_R(f)$, and return the final flow f as their output.

The following lemma is fundamental in understanding the basic strategy behind these algorithms.

Lemma 7.2 *Let f be any flow and f^* a maximum flow in G, and let $G_R(f)$ be the residual graph for f. The value of a maximum flow in $G_R(f)$ is $|f^*| - |f|$.*

***Proof* 7.4** Let f' be any flow in $G_R(f)$. Define $f + f'$ to be the flow defined by the flow function $f(v, w) + f'(v, w)$ for each edge (v, w). Observe that $f + f'$ is a feasible flow in G of value $|f| + |f'|$. Since f^* is the maximum flow possible in G, $|f'| \leq |f^*| - |f|$. Similarly define $f^* - f$ to be a flow in $G_R(f)$ defined by $f^*(v, w) - f(v, w)$ in each edge (v, w), and this is a feasible flow in $G_R(f)$ of value $|f^*| - |f|$, and it is a maximum flow in $G_R(f)$. □

Blocking flow: A flow f is a **blocking flow** if every path in G from s to t contains a saturated edge. It is important to note that a blocking flow is not necessarily a maximum flow. There may be augmenting paths that increase the flow on some edges and decrease the flow on other edges (by increasing the flow in the reverse direction).

Layered networks: Let $G_R(f)$ be the residual graph with respect to a flow f. The level of a vertex v is the length of a shortest path (using the least number of edges) from s to v in $G_R(f)$. The level graph L for f is the subgraph of $G_R(f)$ containing vertices reachable from s and only the edges (v, w) such that $\text{dist}(s, w) = 1 + \text{dist}(s, v)$. L contains all shortest-length augmenting paths and can be constructed in $O(|E|)$ time.

The Maximum Flow algorithm proposed by Dinitz [1970] starts with the zero flow, and iteratively increases the flow by augmenting it with a blocking flow in $G_R(f)$ until t is not reachable from s in $G_R(f)$. At each step the current flow is replaced by the sum of the current flow and the blocking flow. Since in each iteration the shortest distance from s to t in the residual graph increases, and the shortest path from s to t is at most $|V| - 1$, this gives an upper bound on the number of iterations of the algorithm.

An algorithm to find a blocking flow that runs in $O(|V|^2)$ time is described here, and this yields an $O(|V|^3)$ max-flow algorithm. There are a number of $O(|V|^2)$ blocking flow algorithms available [Karzanov 1974, Malhotra et al. 1978, Tarjan 1983], some of which are described in detail in Tarjan [1983].

7.7.9 Blocking Flows

Dinitz's algorithm to find a blocking flow runs in $O(|V||E|)$ time [Dinitz 1970]. The main step is to find paths from the source to the sink and saturate them by pushing as much flow as possible on these paths. Every time the flow is increased by pushing more flow along an augmenting path, one of the edges on this path becomes saturated. It takes $O(|V|)$ time to compute the amount of flow that can be pushed on the path. Since there are $|E|$ edges, this yields an upper bound of $O(|V||E|)$ steps on the running time of the algorithm.

Malhotra–Kumar–Maheshwari Blocking Flow Algorithm. The algorithm has a current flow function f and its corresponding residual graph $G_R(f)$. Define for each node $v \in G_R(f)$, a quantity $tp[v]$ that specifies its maximum throughput, that is, either the sum of the capacities of the incoming arcs or the sum of the capacities of the outgoing arcs, whichever is smaller. $tp[v]$ represents the maximum flow that could pass through v in any feasible blocking flow in the residual graph. Vertices for which the throughput is zero are deleted from $G_R(f)$.

The algorithm selects a vertex u for which its throughput is a minimum among all vertices with nonzero throughput. It then greedily pushes a flow of $tp[u]$ from u toward t, level by level in the layered residual

graph. This can be done by creating a queue, which initially contains u and which is assigned the task of pushing $tp[u]$ out of it. In each step, the vertex v at the front of the queue is removed, and the arcs going out of v are scanned one at a time, and as much flow as possible is pushed out of them until v's allocated flow has been pushed out. For each arc (v, w) that the algorithm pushed flow through, it updates the residual capacity of the arc (v, w) and places w on a queue (if it is not already there) and increments the net incoming flow into w. Also, $tp[v]$ is reduced by the amount of flow that was sent through it now. The flow finally reaches t, and the algorithm never comes across a vertex that has incoming flow that exceeds its outgoing capacity since u was chosen as a vertex with the smallest throughput. The preceding idea is again repeated to pull a flow of $tp[u]$ from the source s to u. Combining the two steps yields a flow of $tp[u]$ from s to t in the residual network that goes through u. The flow f is augmented by this amount. Vertex u is deleted from the residual graph, along with any other vertices that have zero throughput.

This procedure is repeated until all vertices are deleted from the residual graph. The algorithm has a blocking flow at this stage since at least one vertex is saturated in every path from s to t. In the algorithm, whenever an edge is saturated, it may be deleted from the residual graph. Since the algorithm uses a greedy strategy to send flows, at most $O(|E|)$ time is spent when an edge is saturated. When finding flow paths to push $tp[u]$, there are at most n times, one each per vertex, when the algorithm pushes a flow that does not saturate the corresponding edge. After this step, u is deleted from the residual graph. Hence, in $O(|E| + |V|^2) = O(|V|^2)$ steps, the algorithm to compute blocking flows terminates.

Goldberg and Tarjan [1988] proposed a preflow push method that runs in $O(|V||E| \log |V|^2/|E|)$ time without explicitly finding a blocking flow at each step.

7.7.10 Applications of Network Flow

There are numerous applications of the Maximum Flow algorithm in scheduling problems of various kinds. See Ahuja et al. [1993] for further details.

7.8 Tour and Traversal Problems

There are many applications for finding certain kinds of paths and tours in graphs. We briefly discuss some of the basic problems.

The **traveling salesman problem (TSP)** is that of finding a shortest tour that visits all of the vertices in a given graph with weights on the edges. It has received considerable attention in the literature [Lawler et al. 1985]. The problem is known to be computationally intractable (NP-hard). Several heuristics are known to solve practical instances. Considerable progress has also been made for finding optimal solutions for graphs with a few thousand vertices.

One of the first graph-theoretic problems to be studied, the **Euler tour problem** asks for the existence of a closed walk in a given connected graph that traverses each edge exactly once. Euler proved that such a closed walk exists if and only if each vertex has even degree [Gibbons 1985]. Such a graph is known as an **Eulerian graph**. Given an Eulerian graph, a Euler tour in it can be computed using DFS in linear time.

Given an edge-weighted graph, the **Chinese postman problem** is that of finding a shortest closed walk that traverses each edge at least once. Although the problem sounds very similar to the TSP problem, it can be solved optimally in polynomial time by reducing it to the matching problem [Ahuja et al. 1993].

Acknowledgments

Samir Khuller's research is supported by National Science Foundation (NSF) Awards CCR-9820965 and CCR-0113192.

Balaji Raghavachari's research is supported by the National Science Foundation under Grant CCR-9820902.

Defining Terms

Assignment problem: That of finding a perfect matching of maximum (or minimum) total weight.
Augmenting path: An alternating path that can be used to augment (increase) the size of a matching.
Biconnected graph: A graph that cannot be disconnected by the removal of any single vertex.
Bipartite graph: A graph in which the vertex set can be partitioned into two sets X and Y, such that each edge connects a node in X with a node in Y.
Blocking flow: A flow function in which any directed path from s to t contains a saturated edge.
Branching: A spanning tree in a rooted graph, such that the root has a path to each vertex.
Chinese postman problem: Asks for a minimum length tour that traverses each edge at least once.
Connected: A graph in which there is a path between each pair of vertices.
Cycle: A path in which the start and end vertices of the path are identical.
Degree: The number of edges incident to a vertex in a graph.
DFS forest: A rooted forest formed by depth-first search.
Directed acyclic graph: A directed graph with no cycles.
Eulerian graph: A graph that has an Euler tour.
Euler tour problem: Asks for a traversal of the edges that visits each edge exactly once.
Forest: An acyclic graph.
Leaves: Vertices of degree one in a tree.
Matching: A subset of edges that do not share a common vertex.
Minimum spanning tree: A spanning tree of minimum total weight.
Network flow: An assignment of flow values to the edges of a graph that satisfies flow conservation, skew symmetry, and capacity constraints.
Path: An ordered list of edges such that any two consecutive edges are incident to a common vertex.
Perfect matching: A matching in which every node is matched by an edge to another node.
Sparse graph: A graph in which $|E| \ll |V|^2$.
***s–t* cut:** A partitioning of the vertex set into S and T such that $s \in S$ and $t \in T$.
Strongly connected: A directed graph in which there is a directed path in each direction between each pair of vertices.
Topological order: A linear ordering of the edges of a DAG such that every edge in the graph goes from left to right.
Traveling salesman problem: Asks for a minimum length tour of a graph that visits all of the vertices exactly once.
Tree: An acyclic graph with $|V| - 1$ edges.
Walk: An ordered sequence of edges (in which edges could repeat) such that any two consecutive edges are incident to a common vertex.

References

Ahuja, R.K., Magnanti, T., and Orlin, J. 1993. *Network Flows*. Prentice Hall, Upper Saddle River, NJ.

Bellman, R. 1958. On a routing problem. *Q. App. Math.*, 16(1):87–90.

Boruvka, O. 1926. O jistem problemu minimalnim. *Praca Moravske Prirodovedecke Spolecnosti*, 3:37–58 (in Czech).

Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. 2001. *Introduction to Algorithms, second edition*. The MIT Press.

DiBattista, G., Eades, P., Tamassia, R., and Tollis, I. 1994. Annotated bibliography on graph drawing algorithms. *Comput. Geom.: Theory Applic.*, 4:235–282.

Dijkstra, E.W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Dinitz, E.A. 1970. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280.

Elias, P., Feinstein, A., and Shannon, C.E. 1956. Note on maximum flow through a network. *IRE Trans. Inf. Theory*, IT-2:117–119.

Even, S. 1979. *Graph Algorithms*. Computer Science Press, Potomac, MD.

Ford, L.R., Jr. and Fulkerson, D.R. 1956. Maximal flow through a network. *Can. J. Math.*, 8:399–404.

Ford, L.R., Jr. and Fulkerson, D.R. 1962. *Flows in Networks*. Princeton University Press.

Fraenkel, A.S. 1970. Economic traversal of labyrinths. *Math. Mag.*, 43:125–130.

Fredman, M. and Tarjan, R.E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615.

Gabow, H.N., Galil, Z., Spencer, T., and Tarjan, R.E. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122.

Gibbons, A.M. 1985. *Algorithmic Graph Theory*. Cambridge University Press, New York.

Goldberg, A.V. and Tarjan, R.E. 1988. A new approach to the maximum-flow problem. *J. ACM*, 35:921–940.

Hochbaum, D.S., Ed. 1996. *Approximation Algorithms for NP-Hand Problems.* PWS Publishing.

Hopcroft, J.E. and Karp, R.M. 1973. An $n^{2.5}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231.

Hopcroft, J.E. and Tarjan, R.E. 1973. Efficient algorithms for graph manipulation. *Commun. ACM*, 16:372–378.

Jarnik, V. 1930. O jistem problemu minimalnim. *Praca Moravske Prirodovedecke Spolecnosti*, 6:57–63 (in Czech).

Karzanov, A.V. 1974. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.*, 15:434–437.

Kruskal, J.B., 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc.*, 7:48–50.

Kuhn, H.W. 1955. The Hungarian method for the assignment problem. *Nav. Res. Logistics Q.*, 2:83–98.

Lawler, E.L. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston.

Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B. 1985. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, New York.

Lucas, E. 1882. *Recreations Mathematiques*. Paris.

Malhotra, V.M., Kumar, M.P., and Maheshwari, S.N. 1978. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Inf. Process. Lett.*, 7:277–278.

Micali, S. and Vazirani, V.V. 1980. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs, pp. 17–27. In *Proc. 21st Annu. Symp. Found. Comput. Sci.*

Papadimitriou, C.H. and Steiglitz, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Upper Saddle River, NJ.

Prim, R.C. 1957. Shortest connection networks and some generalizations. *Bell Sys. Tech. J.*, 36:1389–1401.

Tarjan, R.E. 1972. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160.

Tarjan, R.E. 1983. *Data Structures and Network Algorithms. SIAM.*

Further Information

The area of graph algorithms continues to be a very active field of research. There are several journals and conferences that discuss advances in the field. Here we name a partial list of some of the important meetings: ACM Symposium on Theory of Computing, IEEE Conference on Foundations of Computer Science, ACM–SIAM Symposium on Discrete Algorithms, the International Colloquium on Automata, Languages and Programming, and the European Symposium on Algorithms. There are many other regional algorithms/theory conferences that carry research papers on graph algorithms. The journals that carry articles on current research in graph algorithms are *Journal of the ACM, SIAM Journal on Computing, SIAM Journal on Discrete Mathematics, Journal of Algorithms, Algorithmica, Journal of Computer and System Sciences, Information and Computation, Information Processing Letters,* and *Theoretical Computer Science.*

To find more details about some of the graph algorithms described in this chapter we refer the reader to the books by Cormen et al. [2001], Even [1979], and Tarjan [1983]. For network flows and matching, a more detailed survey regarding various approaches can be found in Tarjan [1983]. Papadimitriou and Steiglitz [1982] discuss the solution of many combinatorial optimization problems using a primal–dual framework.

Current research on graph algorithms focuses on approximation algorithms [Hochbaum 1996], dynamic algorithms, and in the area of graph layout and drawing [DiBattista et al. 1994].

8

Algebraic Algorithms

Angel Diaz
IBM Research

Erich Kaltófen
North Carolina State University

Victor Y. Pan
Lehman College, CUNY

8.1 Introduction

The title's subject is the algorithmic approach to algebra: arithmetic with numbers, polynomials, matrices, differential polynomials, such as $y'' + (1/2 + x^4/4)y$, truncated series, and algebraic sets, i.e., quantified expressions such as $\exists x \in \mathbb{R} : x^4 + p \cdot x + q = 0$, which describes a subset of the two-dimensional space with coordinates p and q for which the given quartic equation has a real root. Algorithms that manipulate such objects are the backbone of modern symbolic mathematics software such as the Maple and Mathematica systems, to name but two among many useful systems. This chapter restricts itself to algorithms in four areas: linear matrix algebra, root finding of univariate polynomials, solution of systems of nonlinear algebraic equations, and polynomial factorization.

8.2 Matrix Computations and Approximation of Polynomial Zeros

This section covers several major algebraic and numerical problems of scientific and engineering computing that are usually solved numerically, with rounding off or chopping the input and computed values to a fixed number of bits that fit the computer precision (Sections 8.2 and 8.3 are devoted to some fundamental

infinite precision symbolic computations, and within Section 8.2 we comment on the infinite precision techniques for some matrix computations). We also study approximation of polynomial zeros, which is an important, fundamental, as well as very popular subject. In our presentation, we will very briefly list the major subtopics of our huge subject and will give some pointers to the references. We will include brief coverage of the topics of the algorithm design and analysis, regarding the complexity of matrix computation and of approximating polynomial zeros. The reader may find further material on these subjects in the survey articles by Pan [1984a, 1991, 1992a, 1995b] and in the books by Bini and Pan [1994, 1996].

8.2.1 Products of Vectors and Matrices, Convolution of Vectors

An $m \times n$ matrix $A = (a_{i,j},\ i = 0, 1, \ldots, m-1;\ j = 0, 1, \ldots, n-1)$ is a two-dimensional array, whose (i, j) entry is $(A)_{i,j} = a_{i,j}$. A is a column vector of dimension m if $n = 1$ and is a row vector of dimension n if $m = 1$. Transposition, hereafter, indicated by the superscript T, transforms a row vector $\boldsymbol{v}^T = [v_0, \ldots, v_{n-1}]$ into a column vector $\boldsymbol{v} = [v_0, \ldots, v_{n-1}]^T$.

For two vectors, $\boldsymbol{u}^T = (u_0, \ldots, u_{m-1})$ and $\boldsymbol{v}^T = (v_0, \ldots, v_{n-1})^T$, their *outer product* is an $m \times n$ matrix,

$$W = \boldsymbol{u}\boldsymbol{v}^T = [w_{i,j}\ ,\ i = 0, \ldots, m-1;\ j = 0, \ldots, n-1]$$

where $w_{i,j} = u_i v_j$, for all i and j, and their *convolution* vector is said to equal

$$\boldsymbol{w} = \boldsymbol{u} \circ \boldsymbol{v} = (w_0, \ldots, w_{m+n-2})^T, \qquad w_k = \sum_{i=0}^{k} u_i v_{k-i}$$

where $u_i = v_j = 0$, for $i \geq m$, $j \geq n$; in fact, $\boldsymbol{w}$ is the coefficient vector of the product of two polynomials,

$$u(x) = \sum_{i=0}^{m-1} u_i x^i \qquad \text{and} \qquad v(x) = \sum_{i=0}^{n-1} v_i x^i$$

having coefficient vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, respectively.

If $m = n$, the scalar value

$$\boldsymbol{v}^T\boldsymbol{u} = \boldsymbol{u}^T\boldsymbol{v} = u_0 v_0 + u_1 v_1 + \cdots + u_{n-1} v_{n-1} = \sum_{i=0}^{n-1} u_i v_i$$

is called the *inner* (*dot*, or *scalar*) *product* of $\boldsymbol{u}$ and $\boldsymbol{v}$.

The straightforward algorithms compute the inner and outer products of $\boldsymbol{u}$ and $\boldsymbol{v}$ and their convolution vector by using $2n - 1$, mn, and $mn + (m-1)(n-1) = 2mn - m - n + 1$ arithmetic operations (hereafter, referred to as **ops**), respectively.

These upper bounds on the numbers of ops for computing the inner and outer products are sharp, that is, cannot be decreased, for the general pair of the input vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, whereas (see, e.g., Bini and Pan [1994]) one may apply the *fast fourier transform* (FFT) in order to compute the convolution vector $\boldsymbol{u} \circ \boldsymbol{v}$ much faster, for larger m and n; namely, it suffices to use $4.5K \log K + 2K$ ops, for $K = 2^k$, $k = \lceil \log(m + n + 1) \rceil$. (Here and hereafter, all logarithms are binary unless specified otherwise.)

If $A = (a_{i,j})$ and $B = (b_{j,k})$ are $m \times n$ and $n \times p$ matrices, respectively, and $\boldsymbol{v} = (v_k)$ is a p-dimensional vector, then the straightforward algorithms compute the vector

$$\boldsymbol{w} = B\boldsymbol{v} = (w_0, \ldots, w_{n-1})^T, \qquad w_i = \sum_{j=0}^{p-1} b_{i,j} v_j, \quad i = 0, \ldots, n-1$$

by using $(2p - 1)n$ ops (sharp bound), and compute the *matrix product*

$$AB = (w_{i,k}\ ,\ i = 0, \ldots, m-1;\ k = 0, \ldots, p-1)$$

by using $2mnp - mp$ ops, which is $2n^3 - n^2$ if $m = n = p$. The latter upper bound is not sharp: the subroutines for $n \times n$ matrix multiplication on some modern computers, such as CRAY and Connection

Machines, rely on algorithms using $O(n^{2.81})$ ops, and some nonpractical algorithms involve $O(n^{2.376})$ ops [Bini and Pan 1994, Golub and Van Loan 1989].

In the special case, where all of the input entries and components are bounded integers having short binary representation, each of the preceding operations with vectors and matrices can be reduced to a single multiplication of 2 longer integers, by means of the techniques of *binary segmentation* (cf. Pan [1984b, Section 40], Pan [1991], Pan [1992b], or Bini and Pan [1994, Examples 36.1–36.3]).

For an $n \times n$ matrix B and an n-dimensional vector $\boldsymbol{v}$, one may compute the vectors $B^i \boldsymbol{v}$, $i = 1, 2, \ldots, k-1$, which define *Krylov sequence* or *Krylov matrix*

$$[B^i \boldsymbol{v},\ i = 0, 1, \ldots, k-1]$$

used as a basis of several computations. The straightforward algorithm takes on $(2n-1)nk$ ops, which is order n^3 if k is of order n. An alternative algorithm first computes the matrix powers

$$B^2, B^4, B^8, \ldots, B^{2^s}, \quad s = \lceil \log k \rceil - 1$$

and then the products of $n \times n$ matrices B^{2^i} by $n \times 2^i$ matrices, for $i = 0, 1, \ldots, s$,

$$\begin{array}{ll}
B & v \\
B^2 & (v,\ Bv) = (B^2 v,\ B^3 v) \\
B^4 & (v,\ Bv,\ B^2 v,\ B^3 v) = (B^4 v,\ B^5 v,\ B^6 v,\ B^7 v) \\
\vdots &
\end{array}$$

The last step completes the evaluation of the Krylov sequence, which amounts to $2s$ matrix multiplications, for $k = n$, and, therefore, can be performed (in theory) in $O(n^{2.376} \log k)$ ops.

8.2.2 Some Computations Related to Matrix Multiplication

Several fundamental matrix computations can be ultimately reduced to relatively few [that is, to a constant number, or, say, to $O(\log n)$] $n \times n$ matrix multiplications. These computations include the evaluation of det A, the **determinant** of an $n \times n$ matrix A; of its *inverse* A^{-1} (where A is nonsingular, that is, where det $A \neq 0$); of the coefficients of its **characteristic polynomial**, $c_A(x) = \det(xI - A)$, x denoting a scalar variable and I being the $n \times n$ identity matrix, which has ones on its diagonal and zeros elsewhere; of its *minimal polynomial*, $m_A(x)$; of its *rank*, rank A; of the solution vector $\boldsymbol{x} = A^{-1}\boldsymbol{v}$ to a nonsingular *linear system of equations*, $A\boldsymbol{x} = \boldsymbol{v}$; of various *orthogonal* and *triangular factorizations* of A; and of a submatrix of A having the maximal rank, as well as some fundamental computations with singular matrices. Consequently, all of these operations can be performed by using (theoretically) $O(n^{2.376})$ ops (cf. Bini and Pan [1994, Chap. 2]). The idea is to represent the input matrix A as a block matrix and, operating with its blocks (rather than with its entries), to apply fast matrix multiplication algorithms. In practice, due to various other considerations (accounting, in particular, for the overhead constants hidden in the O notation, for the memory space requirements, and particularly, for numerical stability problems), these computations are based either on the straightforward algorithm for matrix multiplication or on other methods allowing order n^3 arithmetic operations (cf. Golub and Van Loan [1989]). Many block matrix algorithms supporting the (nonpractical) estimate $O(n^{2.376})$, however, become practically important for parallel computations (see Section 8.2.7).

In the next six sections, we will more closely consider the solution of a linear system of equations, $A\boldsymbol{v} = \boldsymbol{b}$, which is the most frequent operation in practice of scientific and engineering computing and is highly important theoretically. We will partition the known solution methods depending on whether the coefficient matrix A is *dense and unstructured*, **sparse**, or *dense and* **structured**.

8.2.3 Gaussian Elimination Algorithm

The solution of a nonsingular linear system $A\boldsymbol{x} = \boldsymbol{v}$ uses only about n^2 ops if the system is lower (or upper) triangular, that is, if all subdiagonal (or superdiagonal) entries of A vanish. For example (cf. Pan [1992b]), let $n = 3$,

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 3 \\ -2x_2 - 2x_3 &= -10 \\ -6x_3 &= -18 \end{aligned}$$

Compute $x_3 = 3$ from the last equation, substitute into the previous ones, and arrive at a triangular system of $n - 1 = 2$ equations. In $n - 1$ (in our case, 2) such recursive substitution steps, we compute the solution.

The triangular case is itself important; furthermore, every nonsingular linear system is reduced to two triangular ones by means of *forward elimination* of the variables, which essentially amounts to computing the PLU factorization of the input matrix A, that is, to computing two lower triangular matrices L and U^T (where L has unit values on its diagonal) and a permutation matrix P such that $A = PLU$. [A permutation matrix P is filled with zeros and ones and has exactly one nonzero entry in each row and in each column; in particular, this implies that $P^T = P^{-1}$. $P\boldsymbol{u}$ has the same components as $\boldsymbol{u}$ but written in a distinct (fixed) order, for any vector $\boldsymbol{u}$]. As soon as the latter factorization is available, we may compute $\boldsymbol{x} = A^{-1}\boldsymbol{v}$ by solving two triangular systems, that is, at first, $L\boldsymbol{y} = P^T\boldsymbol{v}$, in $\boldsymbol{y}$, and then $U\boldsymbol{x} = \boldsymbol{y}$, in $\boldsymbol{x}$. Computing the factorization (elimination stage) is more costly than the subsequent *back substitution stage*, the latter involving about $2n^2$ ops. The Gaussian classical algorithm for elimination requires about $2n^3/3$ ops, not counting some comparisons, generally required in order to ensure appropriate *pivoting*, also called *elimination ordering*. Pivoting enables us to avoid divisions by small values, which could have caused numerical stability problems. Theoretically, one may employ fast matrix multiplication and compute the matrices P, L, and U in $O(n^{2.376})$ ops [Aho et al. 1974] [and then compute the vectors $\boldsymbol{y}$ and $\boldsymbol{x}$ in $O(n^2)$ ops]. Pivoting can be dropped for some important classes of linear systems, notably, for *positive definite* and for *diagonally dominant* systems [Golub and Van Loan 1989, Pan 1991, 1992b, Bini and Pan 1994].

We refer the reader to Golub and Van Loan [1989, pp. 82–83], or Pan [1992b, p. 794], on sensitivity of the solution to the input and roundoff errors in numerical computing. The output errors grow with the **condition number** of A, represented by $\|A\|\|A^{-1}\|$ for an appropriate matrix norm or by the ratio of maximum and minimum singular values of A. Except for ill-conditioned linear systems $A\boldsymbol{x} = \boldsymbol{v}$, for which the condition number of A is very large, a rough initial approximation to the solution can be rapidly refined (cf. Golub and Van Loan [1989]) via the *iterative improvement algorithm*, as soon as we know P and rough approximations to the matrices L and U of the PLU factorization of A. Then b correct bits of each output value can be computed in $(b + n)n^2$ ops as $b \to \infty$.

8.2.4 Singular Linear Systems of Equations

If the matrix A is **singular** (in particular, if A is rectangular), then the linear system $A\boldsymbol{x} = \boldsymbol{v}$ is either overdetermined, that is, has no solution, or underdetermined, that is, has infinitely many solution vectors. All of them can be represented as $\{\boldsymbol{x}_0 + \boldsymbol{y}\}$, where $\boldsymbol{x}_0$ is a fixed solution vector and $\boldsymbol{y}$ is a vector from the *null space* of A, $\{\boldsymbol{y} : A\boldsymbol{y} = \boldsymbol{0}\}$, that is, $\boldsymbol{y}$ is a solution of the homogeneous linear system $A\boldsymbol{y} = \boldsymbol{0}$. (The null space of an $n \times n$ matrix A is a linear space of the dimension n–rank A.) A vector $\boldsymbol{x}_0$ and a basis for the null-space of A can be computed by using $O(n^{2.376})$ ops if A is an $n \times n$ matrix or by using $O(mn^{1.736})$ ops if A is an $m \times n$ or $n \times m$ matrix and if $m \geq n$ (cf. Bini and Pan [1994]).

For an overdetermined linear system $A\boldsymbol{x} = \boldsymbol{v}$, having no solution, one may compute a vector $\boldsymbol{x}$ minimizing the norm of the residual vector, $\|\boldsymbol{v} - A\boldsymbol{x}\|$. It is most customary to minimize the Euclidean norm,

$$\|\boldsymbol{u}\| = \left(\sum_i |u_i|^2\right)^{1/2}, \quad \boldsymbol{u} = \boldsymbol{v} - A\boldsymbol{x} = (u_i)$$

This defines a least-squares solution, which is relatively easy to compute both practically and theoretically ($O(n^{2.376})$ ops suffice in theory) (cf. Bini and Pan [1994] and Golub and Van Loan [1989]).

8.2.5 Sparse Linear Systems (Including Banded Systems), Direct and Iterative Solution Algorithms

A matrix is sparse if it is filled mostly with zeros, say, if its all nonzero entries lie on 3 or 5 of its diagonals. In many important applications, in particular, solving partial and ordinary differential equations (PDEs and ODEs), one has to solve linear systems whose matrix is sparse and where, moreover, the disposition of its nonzero entries has a certain structure. Then, memory space and computation time can be dramatically decreased (say, from order n^2 to order $n \log n$ words of memory and from n^3 to $n^{3/2}$ or $n \log n$ ops) by using some special data structures and special solution methods. The methods are either direct, that is, are modifications of Gaussian elimination with some special policies of elimination ordering that preserve sparsity during the computation (notably, *Markowitz rule* and *nested dissection* [George and Liu 1981, Gilbert and Tarjan 1987, Lipton et al. 1979, Pan 1993]), or various iterative algorithms. The latter algorithms rely either on computing Krylov sequences [Saad 1995] or on multilevel or multigrid techniques [McCormick 1987, Pan and Reif 1992], specialized for solving linear systems that arise from discretization of PDEs. An important particular class of sparse linear systems is formed by *banded linear systems* with $n \times n$ coefficient matrices $A = (a_{i,j})$ where $a_{i,j} = 0$ if $i - j > g$ or $j - i > h$, for $g + h$ being much less than n. For banded linear systems, the nested dissection methods are known under the name of *block cyclic reduction* methods and are highly effective, but Pan et al. [1995] give some alternative algorithms, too. Some special techniques for computation of Krylov sequences for sparse and other special matrices A can be found in Pan [1995a]; according to these techniques, Krylov sequence is recovered from the solution of the associated linear system $(I - A)\,\boldsymbol{x} = \boldsymbol{v}$, which is solved fast in the case of a special matrix A.

8.2.6 Dense and Structured Matrices and Linear Systems

Many dense $n \times n$ matrices are defined by $O(n)$, say, by less than $2n$, parameters and can be multiplied by a vector by using $O(n \log n)$ or $O(n \log^2 n)$ ops. Such matrices arise in numerous applications (to signal and image processing, coding, algebraic computation, PDEs, integral equations, particle simulation, Markov chains, and many others). An important example is given by $n \times n$ *Toeplitz matrices* $T = (t_{i,j})$, $t_{i,j} = t_{i+1,j+1}$ for $i, j = 0, 1, \ldots, n-1$. Such a matrix can be represented by $2n-1$ entries of its first row and first column or by $2n - 1$ entries of its first and last columns. The product $T\boldsymbol{v}$ is defined by vector convolution, and its computation uses $O(n \log n)$ ops. Other major examples are given by *Hankel matrices* (obtained by reflecting the row or column sets of Toeplitz matrices), *circulant* (which are a subclass of Toeplitz matrices), and *Bezout*, *Sylvester*, *Vandermonde*, and *Cauchy* matrices. The known solution algorithms for linear systems with such dense structured coefficient matrices use from order $n \log n$ to order $n \log^2 n$ ops. These properties and algorithms are extended via associating some linear operators of displacement and scaling to some more general classes of matrices and linear systems. We refer the reader to Bini and Pan [1994] for many details and further bibliography.

8.2.7 Parallel Matrix Computations

Algorithms for matrix multiplication are particularly suitable for parallel implementation; one may exploit natural association of processors to rows and/or columns of matrices or to their blocks, particularly, in the implementation of matrix multiplication on loosely coupled multiprocessors (cf. Golub and Van Loan [1989] and Quinn [1994]). This motivated particular attention to and rapid progress in devising effective parallel algorithms for block matrix computations. The complexity of parallel computations is usually represented by the computational and communication time and the number of processors involved; decreasing all of these parameters, we face a tradeoff; the product of time and processor bounds (called potential work of parallel algorithms) cannot usually be made substantially smaller than the sequential time bound for the solution. This follows because, according to a variant of *Brent's scheduling principle*, a

single processor can simulate the work of s processors in time $O(s)$. The usual goal of designing a parallel algorithm is in decreasing its parallel time bound (ideally, to a constant, logarithmic or polylogarithmic level, relative to n) and keeping its work bound at the level of the record sequential time bound for the same computational problem (within constant, logarithmic, or at worst polylog factors). This goal has been easily achieved for matrix and vector multiplications, but turned out to be nontrivial for linear system solving, inversion, and some other related computational problems. The recent solution for general matrices [Kaltofen and Pan 1991, 1992] relies on computation of a Krylov sequence and the coefficients of the minimum polynomial of a matrix, by using randomization and auxiliary computations with structured matrices (see the details in Bini and Pan [1994]).

8.2.8 Rational Matrix Computations, Computations in Finite Fields and Semirings

Rational algebraic computations with matrices are performed for a rational input given with no errors, and the computations are also performed with no errors. The precision of computing can be bounded by reducing the computations modulo one or several fixed primes or prime powers. At the end, the exact output values $z = p/q$ are recovered from $z \bmod M$ (if M is sufficiently large relative to p and q) by using the continued fraction approximation algorithm, which is the Euclidean algorithm applied to integers (cf. Pan [1991, 1992a], and Bini and Pan [1994, Section 3 of Chap. 3]). If the output z is known to be an integer lying between $-m$ and m and if $M > 2m$, then z is recovered from $z \bmod M$ as follows:

$$z = \begin{cases} z \bmod M & \text{if } z \bmod M < m \\ -M + z \bmod M & \text{otherwise} \end{cases}$$

The reduction modulo a prime p may turn a nonsingular matrix A and a nonsingular linear system $A\boldsymbol{x} = \boldsymbol{v}$ into singular ones, but this is proved to occur only with a low probability for a random choice of the prime p in a fixed sufficiently large interval (see Bini and Pan [1994, Section 3 of Chap. 4]). To compute the output values z modulo M for a large M, one may first compute them modulo several relatively prime integers $m_1, m_2, \ldots, m_k$ having no common divisors and such that $m_1, m_2, \ldots, m_k > M$ and then easily recover $z \bmod M$ by means of the Chinese remainder algorithm. For matrix and polynomial computations, there is an effective alternative technique of *p-adic* (*Newton–Hensel*) *lifting* (cf. Bini and Pan [1994, Section 3 of Chap. 3]), which is particularly powerful for computations with dense structured matrices, since it preserves the structure of a matrix. We refer the reader to Bareiss [1968] and Geddes et al. [1992] for some special techniques, which enable one to control the growth of all intermediate values computed in the process of performing rational Gaussian elimination, with no roundoff and no reduction modulo an integer.

Gondran and Minoux [1984] and Pan [1993] describe some applications of matrix computations on semirings (with no divisions and subtractions allowed) to graph and combinatorial computations.

8.2.9 Matrix Eigenvalues and Singular Values Problems

The matrix eigenvalue problem is one of the major problems of matrix computation: given an $n \times n$ matrix A, one seeks a $k \times k$ diagonal matrix Λ and an $n \times k$ matrix V of full rank k such that

$$AV = \Lambda V \tag{8.1}$$

The diagonal entries of Λ are called the *eigenvalues* of A; the entry (i, i) of Λ is associated with the ith column of V, called an *eigenvector* of A. The eigenvalues of an $n \times n$ matrix A coincide with the zeros of the characteristic polynomial

$$c_A(x) = \det(xI - A)$$

If this polynomial has n distinct zeros, then $k = n$, and V of Equation 8.1 is a nonsingular $n \times n$ matrix. The matrix $A = I + Z$, where $Z = (z_{i,j})$, $z_{i,j} = 0$ unless $j = i + 1$, $z_{i,i+1} = 1$, is an example of a matrix for which $k = 1$, so that the matrix V degenerates to a vector.

In principle, one may compute the coefficients of $c_A(x)$, the characteristic polynomial of A, and then approximate its zeros (see Section 8.3) in order to approximate the eigenvalues of A. Given the eigenvalues, the corresponding eigenvectors can be recovered by means of the inverse power iteration [Golub and Van Loan 1989, Wilkinson 1965]. Practically, the computation of the eigenvalues via the computation of the coefficients of $c_A(x)$ is not recommended, due to arising numerical stability problems [Wilkinson 1965], and most frequently, the eigenvalues and eigenvectors of a general (unsymmetric) matrix are approximated by means of the *QR algorithm* [Wilkinson 1965, Watkins 1982, Golub and Van Loan 1989]. Before application of this algorithm, the matrix A is simplified by transforming it into the more special (*Hessenberg*) *form* H, by a *similarity transformation*,

$$H = UAU^H \tag{8.2}$$

where $U = (u_{i,j})$ is a unitary matrix, where $U^H U = I$, where $U^H = (\overline{u}_{j,i})$ is the Hermitian transpose of U, with $\overline{z}$ denoting the complex conjugate of z; $U^H = U^T$ if U is a real matrix [Golub and Van Loan 1989]. Similarity transformation into Hessenberg form is one of examples of *rational transformations* of a matrix into special *canonical forms*, of which transformations into *Smith* and *Hermite forms* are two other most important representatives [Kaltofen et al. 1990, Geddes et al. 1992, Giesbrecht 1995].

In practice, the eigenvalue problem is very frequently symmetric, that is, arises for a real symmetric matrix A, for which

$$A^T = (a_{j,i}) = A = (a_{i,j})$$

or for complex Hermitian matrices A, for which

$$A^H = (\overline{a}_{j,i}) = A = (a_{i,j})$$

For real symmetric or Hermitian matrices A, the eigenvalue problem (called symmetric) is treated much more easily than in the unsymmetric case. In particular, in the symmetric case, we have $k = n$, that is, the matrix V of Equation 8.1 is a nonsingular $n \times n$ matrix, and moreover, all of the eigenvalues of A are real and little sensitive to small input perturbations of A (according to the Courant–Fisher minimization criterion [Parlett 1980, Golub and Van Loan 1989]).

Furthermore, similarity transformation of A to the Hessenberg form gives much stronger results in the symmetric case: the original problem is reduced to one for a symmetric tridiagonal matrix H of Equation 8.2 (this can be achieved via the Lanczos algorithm, cf. Golub and Van Loan [1989] or Bini and Pan [1994, Section 3 of Chap. 2]). For such a matrix H, application of the QR algorithm is dramatically simplified; moreover, two competitive algorithms are also widely used, that is, the *bisection* [Parlett 1980] (a slightly slower but very robust algorithm) and the *divide-and-conquer* method [Cuppen 1981, Golub and Van Loan 1989]. The latter method has a modification [Bini and Pan 1991] that only uses $O(n \log^2 n(\log n + \log^2 b))$ arithmetic operations in order to compute all of the eigenvalues of an $n \times n$ symmetric tridiagonal matrix A within the output error bound $2^{-b}\|A\|$, where $\|A\| \le n \max |a_{i,j}|$.

The eigenvalue problem has a generalization, where generalized eigenvalues and eigenvectors for a pair A, B of matrices are sought, such that

$$AV = B\Lambda V$$

(the solution algorithm should proceed without computing the matrix $B^{-1}A$, so as to avoid numerical stability problems).

In another highly important extension of the symmetric eigenvalue problem, one seeks a singular value decomposition (SVD) of a (generally unsymmetric and, possibly, rectangular) matrix A: $A = U\Sigma V^T$, where U and V are unitary matrices, $U^H U = V^H V = I$, and Σ is a diagonal (generally rectangular)

matrix, filled with zeros, except for its diagonal, filled with (positive) singular values of A and possibly, with zeros. The SVD is widely used in the study of numerical stability of matrix computations and in numerical treatment of singular and ill-conditioned (close to singular) matrices. An alternative tool is orthogonal (QR) factorization of a matrix, which is not as refined as SVD but is a little easier to compute [Golub and Van Loan 1989]. The squares of the singular values of A equal the eigenvalues of the Hermitian (or real symmetric) matrix $A^H A$, and the SVD of A can be also easily recovered from the eigenvalue decomposition of the Hermitian matrix

$$\begin{bmatrix} 0 & A^H \\ A & 0 \end{bmatrix}$$

but more popular are some effective direct methods for the computation of the SVD [Golub and Van Loan 1989].

8.2.10 Approximating Polynomial Zeros

Solution of an nth degree polynomial equation,

$$p(x) = \sum_{i=0}^{n} p_i \, x^i = 0, \quad p_n \neq 0$$

(where one may assume that $p_{n-1} = 0$; this can be ensured via shifting the variable x) is a classical problem that has greatly influenced the development of mathematics throughout the centuries [Pan 1995b]. The problem remains highly important for the theory and practice of present day computing, and dozens of new algorithms for its approximate solution appear every year. Among the existent implementations of such algorithms, the practical heuristic champions in efficiency (in terms of computer time and memory space used, according to the results of many experiments) are various modifications of *Newton's iteration*, $z(i+1) = z(i) - a(i)p(z(i))/p'(z(i))$, $a(i)$ being the step-size parameter [Madsen 1973], *Laguerre's method* [Hansen et al. 1977, Foster 1981], and the randomized *Jenkins–Traub algorithm* [1970] [all three for approximating a single zero z of $p(x)$], which can be extended to approximating other zeros by means of deflation of the input polynomial via its numerical division by $x - z$. For simultaneous approximation of all of the zeros of $p(x)$ one may apply the Durand–Kerner algorithm, which is defined by the following recurrence:

$$z_j(i+1) = \frac{z_j(i) - p((z_j(i))}{z_j(i) - z_k(i)}, \quad j = 1, \ldots, n, \quad i = 1, 2, \ldots \tag{8.3}$$

Here, the customary choice for the n initial approximations $z_j(0)$ to the n zeros of

$$p(x) = p_n \prod_{j=1}^{n} (x - z_j)$$

is given by $z_j(0) = Z \exp(2\pi\sqrt{-1}/n)$, $j = 1, \ldots, n$, with Z exceeding (by some fixed factor $t > 1$) $\max_j |z_j|$; for instance, one may set

$$Z = 2t \max_{i<n}(p_i/p_n) \tag{8.4}$$

For a fixed i and for all j, the computation according to Equation 8.3 is simple, only involving order n^2 ops, and according to the results of many experiments, the iteration Equation 8.3 rapidly converges to the solution, though no theory confirms or explains these results. Similar is the situation with various

modifications of this algorithm, which are now even more popular than the original algorithms and many of which are listed in Pan [1992a, 1992b] (also cf. Bini and Pan [1996] and McNamee [1993]).

On the other hand, there are two groups of algorithms that, when implemented, promise to be competitive or even substantially superior to Newton's and Laguerre's iteration, the algorithm by Jenkins and Traub, and all of the algorithms of the Durand–Kerner type. One such group is given by the modern modifications and improvements (due to Pan [1987, 1994a, 1994b] and Renegar [1989]) of *Weyl's quadtree construction* of 1924. In this approach, an initial square S, containing all the zeros of $p(x)$ [say, $S = \{x, |Im\ x| < Z, |Re\ x| < Z\}$ for Z of Eq. (8.4)], is recursively partitioned into four congruent subsquares. In the center of each of them, a proximity test is applied that estimates the distance from this center to the closest zero of $p(x)$. If such a distance exceeds one-half of the diagonal length, then the subsquare contains no zeros of $p(x)$ and is discarded. When this process ensures a strong isolation from each other for the components formed by the remaining squares, then certain extensions of Newton's iteration [Renegar 1989, Pan 1994a, 1994b], or some iterative techniques based on numerical integration [Pan 1987] are applied and very rapidly converge to the desired approximations to the zeros of $p(x)$, within the error bound $2^{-b}Z$ for Z of Equation 8.4. As a result, the algorithms of Pan [1987, 1994a, 1994b] solve the entire problem of approximating (within $2^{-b}Z$) all of the zeros of $p(x)$ at the overall cost of performing $O((n^2 \log n) \log(bn))$ ops (cf. Bini and Pan [1996]), versus order n^2 operations at each iteration of Durand–Kerner type.

The second group is given by the divide-and-conquer algorithms. They first compute a sufficiently wide annulus A, which is free of the zeros of $p(x)$ and contains comparable numbers of such zeros (that is, the same numbers up to a fixed constant factor) in its exterior and its interior. Then the two factors of $p(x)$ are numerically computed, that is, $F(x)$ having all its zeros in the interior of the annulus, and $G(x) = p(x)/F(x)$ having no zeros there. The same process is recursively repeated for $F(x)$ and $G(x)$ until factorization of $p(x)$ into the product of linear factors is computed numerically. From this factorization, approximations to all of the zeros of $p(x)$ are obtained. The algorithms of Pan [1995a, 1996] based on this approach only require $O(n \log(bn) (\log n)^2)$ ops in order to approximate all of the n zeros of $p(x)$ within $2^{-b}Z$ for Z of Eq. (8.4). (Note that this is a quite sharp bound: at least n ops are necessary in order to output n distinct values.)

The computations for the polynomial zero problem are ill conditioned, that is, they generally require a high precision for the worst-case input polynomials in order to ensure a required output precision, no matter which algorithm is applied for the solution. Consider, for instance, the polynomial $(x - \frac{6}{7})^n$ and perturb its x-free coefficient by 2^{-bn}. Observe the resulting jumps of the zero $x = 6/7$ by 2^{-b}, and observe similar jumps if the coefficients p_i are perturbed by $2^{(i-n)b}$ for $i = 1, 2, \ldots, n-1$. Therefore, to ensure the output precision of b bits, we need an input precision of at least $(n - i)b$ bits for each coefficient $p_i, i = 0, 1, \ldots, n-1$. Consequently, for the worst-case input polynomial $p(x)$, any solution algorithm needs at least about a factor n increase of the precision of the input and of computing versus the output precision.

Numerically unstable algorithms may require even a higher input and computation precision, but inspection shows that this is not the case for the algorithms of Pan [1987, 1994a, 1994b, 1995a, 1996] and Renegar [1989] (cf. Bini and Pan [1996]).

8.2.11 Fast Fourier Transform and Fast Polynomial Arithmetic

To yield the record complexity bounds for approximating polynomial zeros, one should exploit fast algorithms for basic operations with polynomials (their multiplication, division, and transformation under the shift of the variable), as well as FFT, both directly and for supporting the fast polynomial arithmetic. The FFT and fast basic polynomial algorithms (including those for multipoint polynomial evaluation and interpolation) are the basis for many other fast polynomial computations, performed both numerically and symbolically (compare the next sections). These basic algorithms, their impact on the field of algebraic computation, and their complexity estimates have been extensively studied in Aho et al. [1974], Borodin and Munro [1975], and Bini and Pan [1994].

8.3 Systems of Nonlinear Equations and Other Applications

Given a system $\{p_1(x_1, \ldots, x_n), p_2(x_1, \ldots, x_n), \ldots, p_r(x_1, \ldots, x_n)\}$ of nonlinear polynomials with rational coefficients [each $p_i(x_1, \ldots, x_n)$ is said to be an element of $\mathbb{Q}[x_1, \ldots, x_n]$, the ring of polynomials in $x_1, \ldots, x_n$ over the field $\mathbb{Q}$ of rational numbers], the n-tuple of complex numbers $(a_1, \ldots, a_n)$ is a common solution of the system, if $f_i(a_1, \ldots, a_n) = 0$ for each i with $1 \leq i \leq r$. In this section, we explore the problem of exactly solving a system of nonlinear equations over the field $\mathbb{Q}$. We provide an overview and cite references to different symbolic techniques used for solving systems of algebraic (polynomial) equations. In particular, we describe methods involving *resultant* and *Gröbner basis* computations.

The *Sylvester resultant method* is the technique most frequently utilized for determining a common zero of two polynomial equations in one variable [Knuth 1981]. However, using the Sylvester method successively to solve a system of multivariate polynomials proves to be inefficient. Successive resultant techniques, in general, lack efficiency as a result of their sensitivity to the ordering of the variables [Kapur and Lakshman 1992]. It is more efficient to eliminate all variables together from a set of polynomials, thus leading to the notion of the *multivariate resultant*. The three most commonly used multivariate resultant formulations are the *Dixon* [Dixon 1908, Kapur and Saxena 1995], *Macaulay* [Macaulay 1916, Canny 1990, Kaltofen and Lakshman 1988], and *sparse resultant formulations* [Canny and Emiris 1993a, Sturmfels 1991].

The theory of Gröbner bases provides powerful tools for performing computations in multivariate polynomial rings. Formulating the problem of solving systems of polynomial equations in terms of polynomial ideals, we will see that a Gröbner basis can be computed from the input polynomial set, thus allowing for a form of back substitution (cf. Section 8.2) in order to compute the common roots.

Although not discussed, it should be noted that the *characteristic set algorithm* can be utilized for polynomial system solving. Ritt [1950] introduced the concept of a characteristic set as a tool for studying solutions of algebraic differential equations. Wu [1984, 1986], in search of an effective method for automatic theorem proving, converted Ritt's method to ordinary polynomial rings. Given the before mentioned system P, the characteristic set algorithm transforms P into a triangular form, such that the set of common zeros of P is equivalent to the set of roots of the triangular system [Kapur and Lakshman 1992].

Throughout this exposition we will also see that these techniques used to solve nonlinear equations can be applied to other problems as well, such as computer-aided design and automatic geometric theorem proving.

8.3.1 Resultant Methods

The question of whether two polynomials $f(x), g(x) \in \mathbb{Q}[x]$,

$$f(x) = f_n x^n + f_{n-1} x^{n-1} + \cdots + f_1 x + f_0$$
$$g(x) = g_m x^m + g_{m-1} x^{m-1} + \cdots + g_1 x + g_0$$

have a common root leads to a condition that has to be satisfied by the coefficients of both f and g. Using a derivation of this condition due to Euler, the *Sylvester matrix* of f and g (which is of order $m + n$) can be formulated. The vanishing of the determinant of the Sylvester matrix, known as the *Sylvester resultant*, is a necessary and sufficient condition for f and g to have common roots [Knuth 1981].

As a running example let us consider the following system in two variables provided by Lazard [1981]:

$$f = x^2 + xy + 2x + y - 1 = 0$$
$$g = x^2 + 3x - y^2 + 2y - 1 = 0$$

The Sylvester resultant can be used as a tool for eliminating several variables from a set of equations [Kapur and Lakshman 1992]. Without loss of generality, the roots of the Sylvester resultant of f and g treated as polynomials in y, whose coefficients are polynomials in x, are the x-coordinates of the common zeros of

f and g. More specifically, the Sylvester resultant of the Lazard system with respect to y is given by the following determinant:

$$\det\left(\begin{bmatrix} x+1 & x^2+2x-1 & 0 \\ 0 & x+1 & x^2+2x-1 \\ -1 & 2 & x^2+3x-1 \end{bmatrix}\right) = -x^3-2x^2+3x$$

The roots of the Sylvester resultant of f and g are $\{-3, 0, 1\}$. For each x value, one can substitute the x value back into the original polynomials yielding the solutions $(-3, 1), (0, 1), (1, -1)$.

The method just outlined can be extended recursively, using *polynomial GCD computations*, to a larger set of multivariate polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$. This technique, however, is impractical for eliminating many variables, due to an explosive growth of the degrees of the polynomials generated in each elimination step.

The Sylvester formulations have led to a *subresultant theory*, developed simultaneously by G. E. Collins and W. S. Brown and J. Traub. The subresultant theory produced an efficient algorithm for computing polynomial GCDs and their resultants, while controlling intermediate expression swell [Brown 1971, Brown and Traub 1971, Collins 1967, 1971, Knuth 1981].

It should be noted that by adopting an implicit representation for symbolic objects, the intermediate expression swell introduced in many symbolic computations can be palliated. Recently, polynomial GCD algorithms have been developed that use implicit representations and thus avoid the computationally costly content and primitive part computations needed in those GCD algorithms for polynomials in explicit representation [Díaz and Kaltofen 1995, Kaltofen 1988, Kaltofen and Trager 1990].

The solvability of a set of nonlinear multivariate polynomials over the field $\mathbb{Q}$ can be determined by the vanishing of a generalization of the Sylvester resultant of two polynomials in a single variable.

Due to the special structure of the Sylvester matrix, Bézout developed a method for computing the resultant as a determinant of order $\max(m, n)$ during the 18th century. Cayley [1865] reformulated Bézout's method leading to Dixon's [1908] extension to the bivariate case. Dixon's method can be generalized to a set

$$\{p_1(x_1, \ldots, x_n), p_2(x_1, \ldots, x_n), \ldots, p_{n+1}(x_1, \ldots, x_n)\}$$

of $n+1$ generic n-degree polynomials in n variables [Kapur et al. 1994]. The vanishing of the Dixon resultant is a necessary and sufficient condition for the polynomials to have a nontrivial projective common zero, and also a necessary condition for the existence of an affine common zero. The Dixon formulation gives the resultant up to a multiple, and hence in the affine case it may happen that the vanishing of the Dixon resultant does not necessarily indicate that the equations in question have a common root. A nontrivial multiple, known as the *projection operator*, can be extracted via a method based on so-called *rank subdeterminant computation* (RSC) [Kapur et al. 1994]. It should be noted that the RSC method can also be applied to the Macaulay and sparse resultant formulations as is detailed here.

In 1916, Macaulay constructed a resultant for n homogeneous polynomials in n variables, which simultaneously generalizes the Sylvester resultant and the determinant of a system of linear equations [Canny et al. 1989, Kapur and Lakshman 1992]. Like the Dixon formulation, the Macaulay resultant is a multiple of the resultant (except in the case of generic homogeneous polynomials, where it produces the exact resultant). For the Macaulay formulation, Canny [1990] has invented a general method that perturbs any polynomial system and extracts a nontrivial projection operator.

Using recent results pertaining to sparse polynomial systems [Gelfand et al. 1994, Sturmfels 1991, Sturmfels and Zelevinsky 1992], the mixed sparse resultant of a system of $n+1$ sparse polynomials in n variables in its matrix form was given by Canny and Emiris [1993a] and consequently improved in Canny and Emiris [1993b, 1994]. Here, sparsity denotes that only certain monomials in each of the $n+1$ polynomials have nonzero coefficients. The determinant of the sparse resultant matrix, such as the Macaulay and Dixon matrices, only yields a projection operation, not the exact resultant.

Suppose we are asked to find the common zeros of a set of n polynomials in n variables $\{p_1(x_1, \ldots, x_n), p_2(x_1, \ldots, x_n), \ldots, p_n(x_1, \ldots, x_n)\}$. By augmenting the polynomial set by a generic linear form [Canny 1990, Canny and Manocha 1991, Kapur and Lakshman 1992], one can construct the *u-resultant* of a given system of polynomials. The u-resultant factors into linear factors over the complex numbers, providing

the common zeros of the given polynomials equations. The u-resultant method takes advantage of the properties of the multivariate resultant, and hence can be constructed using either Dixon's, Macaulay's, or sparse formulations.

Consider the previous example augmented by a generic linear form

$$f_1 = x^2 + xy + 2x + y - 1 = 0$$
$$f_2 = x^2 + 3x - y^2 + 2y - 1 = 0$$
$$f_1 = ux + vy + w = 0$$

As described in Canny et al. [1989], the following matrix M corresponds to the Macaulay u-resultant of the preceding system of polynomials, with z being the homogenizing variable:

$$M = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & u & 0 & 0 & 0 \\ 2 & 0 & 1 & 3 & 0 & 1 & 0 & u & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & v & 0 & 0 & 0 \\ 1 & 2 & 1 & 2 & 3 & 0 & w & v & u & 0 \\ -1 & 0 & 2 & -1 & 0 & 3 & 0 & w & 0 & u \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & -1 & 0 & 0 & v & 0 \\ 0 & -1 & 1 & 0 & -1 & 2 & 0 & 0 & w & v \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & w \end{bmatrix}$$

It should be noted that

$$\det(M) = (u - v + w)(-3u + v + w)(v + w)(u - v)$$

corresponds to the affine solutions $(1, -1)$, $(-3, 1)$, $(0, 1)$, and one solution at infinity. An empirical comparison of the detailed resultant formulations can be found in Kapur and Saxena [1995]. Recently, the multivariate resultant formulations are being used for other applications such as *algebraic and geometric reasoning* [Kapur et al. 1994], *computer-aided design* [Stederberg and Goldman 1986], and for *implicitization and finding base points* [Chionh 1990].

8.3.2 Gröbner Bases

Solving systems of nonlinear equations can be formulated in terms of polynomial ideals [Becker and Weispfenning 1993, Geddes et al. 1992, Winkler 1996]. Let us first establish some terminology.

The ideal generated by a system of polynomial equations $p_1, \ldots, p_r$ over $\mathbb{Q}[x_1, \ldots, x_n]$ is the set of all linear combinations

$$(p_1, \ldots, p_r) = \{h_1 p_1 + \cdots + h_r p_r \mid h_1, \ldots, h_r \in \mathbb{Q}[x_1, \ldots, x_n]\}$$

The algebraic variety of $p_1, \ldots, p_r \in \mathbb{Q}[x_1, \ldots, x_n]$ is the set of their common zeros,

$$V(p_1, \ldots, p_r) = \{(a_1, \ldots, a_n) \in \mathbb{C}^n \mid f_1(a_1, \ldots, a_n) = \cdots = f_r(a_1, \ldots, a_n) = 0\}$$

A version of the *Hilbert Nullstellensatz* states that

$$V(p_1, \ldots, p_r) = \text{the empty set } \emptyset \quad \Longleftrightarrow \quad 1 \in (p_1, \ldots, p_r) \text{ over } \mathbb{Q}[x_1, \ldots, x_n]$$

which relates the solvability of polynomial systems to the ideal membership problem.

A term $t = x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}$ of a polynomial is a product of powers with $\deg(t) = e_1 + e_2 + \cdots + e_n$. In order to add needed structure to the polynomial ring we will require that the terms in a polynomial be ordered in an admissible fashion [Geddes et al. 1992, Kapur and Lakshman 1992]. Two of the most common admissible orderings are the **lexicographic order** ($\prec_l$), where terms are ordered as in a dictionary, and the **degree order**

($\prec_d$), where terms are first compared by their degrees with equal degree terms compared lexicographically. A variation to the lexicographic order is the *reverse lexicographic order*, where the lexicographic order is reversed [Davenport et al. 1988, p. 96].

It is this previously mentioned structure that permits a type of simplification known as polynomial reduction. Much like a polynomial remainder process, the process of polynomial reduction involves subtracting a multiple of one polynomial from another to obtain a smaller degree result [Becker and Weispfenning 1993, Geddes et al. 1992, Kapur and Lakshman 1992, Winkler 1996].

A polynomial g is said to be reducible with respect to a set $P = \{p_1, \ldots, p_r\}$ of polynomials if it can be reduced by one or more polynomials in P. When g is no longer reducible by the polynomials in P, we say that g is *reduced* or is *a normal form* with respect to P.

For an arbitrary set of basis polynomials, it is possible that different reduction sequences applied to a given polynomial g could reduce to different normal forms. A basis $G \subseteq \mathbb{Q}[x_1, \ldots, x_n]$ is a *Gröbner basis* if and only if every polynomial in $\mathbb{Q}[x_1, \ldots, x_n]$ has a unique normal form with respect to G. Buchberger [1965, 1976, 1983, 1985] showed that every basis for an ideal $(p_1, \ldots, p_r)$ in $\mathbb{Q}[x_1, \ldots, x_n]$ can be converted into a Gröbner basis $\{p_1^*, \ldots, p_s^*\} = GB(p_1, \ldots, p_r)$, concomitantly designing an algorithm that transforms an arbitrary ideal basis into a Gröbner basis. Another characteristic of Gröbner bases is that by using the previously mentioned reduction process we have

$$g \in (p_1, \ldots, p_r) \quad \Longleftrightarrow \quad (g \bmod p_1^*, \ldots, p_s^*) = 0$$

Further, by using the Nullstellensatz it can be shown that $p_1, \ldots, p_r$ viewed as a system of algebraic equations is solvable if and only if $1 \notin GB(p_1, \ldots, p_r)$.

Depending on which admissible term ordering is used in the Gröbner bases construction, an ideal can have different Gröbner bases. However, an ideal cannot have different (reduced) Gröbner bases for the same term ordering.

Any system of polynomial equations can be solved using a lexicographic Gröbner basis for the ideal generated by the given polynomials. It has been observed, however, that Gröbner bases, more specifically lexicographic Gröbner bases, are hard to compute [Becker and Weispfenning 1993, Geddes et al. 1992, Lakshman 1990, Winkler 1996]. In the case of zero-dimensional ideals, those whose varieties have only isolated points, Faugère, et al. [1993] outlined a change of basis algorithm which can be utilized for solving zero-dimensional systems of equations. In the zero-dimensional case, one computes a Gröbner basis for the ideal generated by a system of polynomials under a degree ordering. The so-called *change of basis algorithm* can then be applied to the degree ordered Gröbner basis to obtain a Gröbner basis under a lexicographic ordering.

Turning to Lazard's example in the form of a polynomial basis,

$$f_1 = x^2 + xy + 2x + y - 1$$
$$f_2 = x^2 + 3x - y^2 + 2y - 1$$

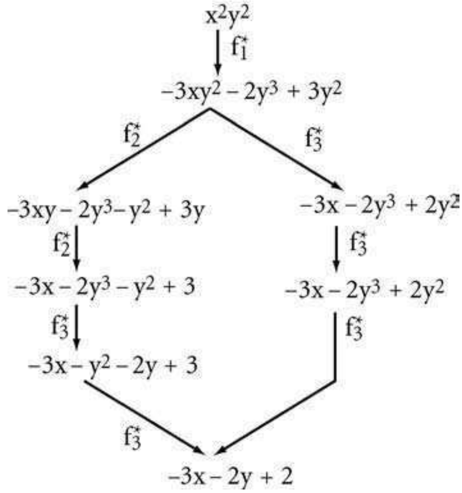
one obtains (under lexicographical ordering with $x \prec_l y$) a Gröbner basis in which the variables are triangularized such that the finitely many solutions can be computed via back substitution:

$$f_1^* = x^2 + 3x + 2y - 2$$
$$f_2^* = xy - x - y + 1$$
$$f_3^* = y^2 - 1$$

It should be noted that the final univariate polynomial is of minimal degree and the polynomials used in the back substitution will have degree no larger than the number of roots.

As an example of the process of polynomial reduction with respect to a Gröbner basis, the following demonstrates two possible reduction sequences to the same normal form. The polynomial x^2y^2 is reduced

with respect to the previously computed Gröbner basis $\{f_1^*, f_2^*, f_3^*\} = GB(f_1, f_2)$ along the following two distinct reduction paths, both yielding $-3x - 2y + 2$ as the normal form.



There is a strong connection between lexicographic Gröbner bases and the previously mentioned resultant techniques. For some types of input polynomials, the computation of a reduced system via resultants might be much faster than the computation of a lexicographic Gröbner basis. A good comparison between the Gröbner computations and the different resultant formulations can be found in Kapur and Saxena [1995].

In a survey article, Buchberger [1985] detailed how Gröbner bases can be used as a tool for many polynomial ideal theoretic operations. Other applications of Gröbner basis computations include automatic geometric theorem proving [Kapur 1986, Wu 1984, 1986], multivariate polynomial factorization and GCD computations [Gianni and Trager 1985], and polynomial interpolation [Lakshman and Saunders 1994, 1995].

8.4 Polynomial Factorization

The problem of factoring polynomials is a fundamental task in symbolic algebra. An example in one's early mathematical education is the factorization $x^2 - y^2 = (x + y) \cdot (x - y)$, which in algebraic terms is a factorization of a polynomial in two variables with integer coefficients. Technology has advanced to a state where most polynomial factorization problems are doable on a computer, in particular, with any of the popular mathematical software, such as the Mathematica or Maple systems. For instance, the factorization of the determinant of a 6×6 symmetric Toeplitz matrix over the integers is computed in Maple as

```
> readlib(showtime) :
> showtime() :
O1 := T := linalg[toeplitz]([a, b, c, d, e, f]);
```

$$T := \begin{bmatrix} a & b & c & d & e & f \\ b & a & b & c & d & e \\ c & b & a & b & c & d \\ d & c & b & a & b & c \\ e & d & c & b & a & b \\ f & e & d & c & b & a \end{bmatrix}$$

```
time 0.03 words7701
O2 := factor(linalg[det](T));
```

$$
\begin{aligned}
&-(2dca - 2bce + 2c^2a - a^3 - da^2 + 2d^2c + d^2a + b^3 + 2abc - 2c^2b \\
&\quad + d^3 + 2ab^2 - 2dcb - 2cb^2 - 2ec^2 + 2eb^2 + 2fcb + 2bae \\
&\quad + b^2f + c^2f + be^2 - ba^2 - fdb - fda - fa^2 - fba + e^2a - 2db^2 \\
&\quad + dc^2 - 2deb - 2dec - dba)(2dca - 2bce - 2c^2a + a^3 \\
&\quad - da^2 - 2d^2c - d^2a + b^3 + 2abc - 2c^2b + d^3 - 2ab^2 + 2dcb \\
&\quad + 2cb^2 + 2ec^2 - 2eb^2 - 2fcb + 2bae + b^2f + c^2f + be^2 - ba^2 \\
&\quad - fdb + fda - fa^2 + fba - e^2a - 2db^2 + dc^2 + 2deb - 2dec \\
&\quad + dba)
\end{aligned}
$$

```
time 27.30 words 857700
```

Clearly, the Toeplitz determinant factorization requires more than tricks from high school algebra. Indeed, the development of modern algorithms for the polynomial factorization problem is one of the great successes of the discipline of symbolic mathematical computation. Kaltofen [1982, 1990, 1992] has surveyed the algorithms until 1992, mostly from a computer science perspective. In this chapter we shall focus on the applications of the known fast methods to problems in science and engineering. For a more extensive set of references, please refer to Kaltofen's survey articles.

8.4.1 Polynomials in a Single Variable over a Finite Field

At first glance, the problem of factoring an integer polynomial modulo a prime number appears to be very similar to the problem of factoring an integer represented in a prime radix. That is simply not so. The factorization of the polynomial $x^{511} - 1$ can be done modulo 2 on a computer in a matter of milliseconds, whereas the factorization of the integer $2^{511} - 1$ into its integer factors is a computational challenge. For those interested: the largest prime factors of $2^{511} - 1$ have 57 and 67 decimals digits, respectively, which makes a tough but not undoable 123 digit product for the number field sieve factorizer [Leyland 1995]. Irreducible factors of polynomials modulo 2 are needed to construct finite fields. For example, the factor $x^9 + x^4 + 1$ of $x^{511} - 1$ leads to a model of the finite field with 2^9 elements, GF(2^9), by simply computing with the polynomial remainders modulo $x^9 + x^4 + 1$ as the elements. Such irreducible polynomials are used for setting up error-correcting codes, such as the BCH codes [MacWilliams and Sloan 1977]. Berlekamp's [1967, 1970] pioneering work on factoring polynomials over a finite field by linear algebra is done with this motivation. The linear algebra tools that Berlekamp used seem to have been introduced to the subject as early as in 1937 by Petr (cf. Št. Schwarz [1956]).

Today, factoring algorithms for univariate polynomials over finite fields form the innermost subalgorithm to lifting-based algorithms for factoring polynomials in one [Zassenhaus 1969] and many [Musser 1975] variables over the integers. When Maple computed the factorization of the previous Toeplitz determinant, it began with factoring a univariate polynomial modulo a prime integer. The case when the prime integer is very large has led to a significant development in computer science itself. As it turns out, by selecting random residues the expected performance of the algorithms can be speeded up exponentially [Berlekamp 1970, Rabin 1980]. Randomization is now an important tool for designing efficient algorithms and has proliferated to many fields of computer science. Paradoxically, the random elements are produced by a congruential random number generator, and the actual computer implementations are quite deterministic, which leads some computer scientists to believe that random bits can be eliminated in general at no exponential slow down. Nonetheless, for the polynomial factoring problem modulo a large prime, no fast methods are known to date that would work without this *probabilistic* approach.

One can measure the computing time of selected algorithms in terms of n, the degree of the input polynomial, and p, the cardinality of the field. When counting arithmetic operations modulo p (including reciprocals), the best known algorithms are quite recent. Berlekamp's 1970 method performs

$O(n^{\omega} + n^{1+o(1)} \log p)$ residue operations. Here and subsequently, ω denotes the exponent implied by the used linear system solver, i.e., $\omega = 3$ when classical methods are used, and $\omega = 2.376$ when asymptotically fast (though impractical) matrix multiplication is assumed. The correction term $o(1)$ accounts for the $\log n$ factors derived from the FFT-based fast polynomial multiplication and remaindering algorithms. An approach in the spirit of Berlekamp's but possibly more practical for $p = 2$ has recently been discovered by Niederreiter [1994]. A very different technique by Cantor and Zassenhaus [1981] first separates factors of different degrees and then splits the resulting polynomials of equal degree factors. It has $O(n^{2+o(1)} \log p)$ complexity and is the basis for the following two methods. Algorithms by von zur Gathen and Shoup [1992] have running time $O(n^{2+o(1)} + n^{1+o(1)} \log p)$ and those by Kaltofen and Shoup [1995] have running time $O(n^{1.815} \log p)$, the latter with fast matrix multiplication.

For n and p simultaneously large, a variant of the method by Kaltofen and Shoup [1995] that uses classical linear algebra and runs in $O(n^{2.5} + n^{1+o(1)} \log p)$ residue operations is the current champion among the practical algorithms. With it Shoup [1996], using his own fast polynomial arithmetic package, has factored a randomlike polynomial of degree 2048 modulo a 2048-bit prime number in about 12 days on a Sparc-10 computer using 68 megabyte of main memory. For even larger n, but smaller p, parallelization helps, and Kaltofen and Lobo [1994] could factor a polynomial of degree $n = 15\,001$ modulo $p = 127$ in about 6 days on 8 computers that are rated at 86.1 MIPS. At the time of this writing, the largest polynomial factored modulo 2 is $X^{216\,091} + X + 1$; this was accomplished by Peter Montgomery in 1991 by using Cantor's fast polynomial multiplication algorithm based on additive transforms [Cantor 1989].

8.4.2 Polynomials in a Single Variable over Fields of Characteristic Zero

As mentioned before, generally usable methods for factoring univariate polynomials over the rational numbers begin with the Hensel lifting techniques introduced by Zassenhaus [1969]. The input polynomial is first factored modulo a suitable prime integer p, and then the factorization is lifted to one modulo p^k for an exponent k of sufficient size to accommodate all possible integer coefficients that any factors of the polynomial might have. The lifting approach is fast in practice, but there are hard-to-factor polynomials on which it runs an exponential time in the degree of the input. This slowdown is due to so-called parasitic modular factors. The polynomial $x^4 + 1$, for example, factors modulo all prime integers but is irreducible over the integers: it is the cyclotomic equation for eighth roots of unity. The products of all subsets of modular factors are candidates for integer factors, and irreducible integer polynomials with exponentially many such subsets exist [Kaltofen et al. 1983].

The elimination of the exponential bottleneck by giving a polynomial-time solution to the integer polynomial factoring problem, due to Lenstra et al. [1982] is considered a major result in computer science algorithm design. The key ingredient to their solution is the construction of integer relations to real or complex numbers. For the simple demonstration of this idea, consider the polynomial

$$x^4 + 2x^3 - 6x^2 - 4x + 8$$

A root of this polynomial is $\alpha \approx 1.236067977$, and $\alpha^2 \approx 1.527864045$. We note that $2\alpha + \alpha^2 \approx 4.000000000$, hence $x^2 + 2x - 4$ is a factor. The main difficulty is to efficiently compute the integer linear relation with relatively small coefficients for the high-precision big-float approximations of the powers of a root. Lenstra et al. [1982] solve this diophantine optimization problem by means of their now famous lattice reduction procedure, which is somewhat reminiscent of the ellipsoid method for linear programming.

The determination of linear integer relations among a set of real or complex numbers is a useful task in science in general. Very recently, some stunning identities could be produced by this method, including the following formula for π [Finch 1995]:

$$\pi = \sum_{n=0}^{\infty} \frac{1}{16^n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right)$$

Even more surprising, the lattice reduction algorithm can prove that no linear integer relation with integers smaller than a chosen parameter exists among the real or complex numbers. There is an efficient alternative to the lattice reduction algorithm, originally due to Ferguson and Forcade [1982] and recently improved by Ferguson and Bailey.

The complexity of factoring an integer polynomial of degree n with coefficients of no more than l bits is thus a polynomial in n and l. From a theoretical point of view, an algorithm with a low estimate is by Miller [1992] and has a running time of $O(n^{5+o(1)}l^{1+o(1)} + n^{4+o(1)}l^{2+o(1)})$ bit operations. It is expected that the relation-finding methods will become usable in practice on hard-to-factor polynomials in the near future. If the hard-to-factor input polynomial is irreducible, an alternate approach can be used to prove its irreducibility. One finds an integer evaluation point at which the integral value of the polynomial has a large prime factor, and the irreducibility follows by mathematical theorems. Monagan [1992] has proven large hard-to-factor polynomials irreducible in this way, which would be hopeless by the lifting algorithm.

Coefficient fields other than finite fields and the rational numbers are of interest. Computing the factorizations of univariate polynomials over the complex numbers is the root finding problem described in the earlier section Approximating Polynomial Zeros. When the coefficient field has an extra variable, such as the field of fractions of polynomials (rational functions) the problem reduces, by an old theorem of Gauss, to factoring multivariate polynomials, which we discuss subsequently. When the coefficient field is the field of Laurent series in t with a finite segment of negative powers,

$$\frac{c_{-k}}{t^k} + \frac{c_{-k+1}}{t^{k-1}} + \cdots + \frac{c_{-1}}{t} + c_0 + c_1 t + c_2 t^2 + \cdots, \quad \text{where } k \geq 0$$

fast methods appeal to the theory of Puiseux series, which constitute the domain of algebraic functions [Walsh 1993].

8.4.3 Polynomials in Two Variables

Factoring bivariate polynomials by reduction to univariate factorization via homomorphic projection and subsequent lifting can be done similarly to the univariate algorithm [Musser 1975]. The second variable y takes the role of the prime integer p and $f(x, y) \bmod y = f(x, 0)$. Lifting is possible only if $f(x, 0)$ had no multiple root. Provided that $f(x, y)$ has no multiple factor, which can be ensured by a simple GCD computation, the *squarefreeness* of $f(x, 0)$ can be obtained by variable translation $\hat{y} = y + a$, where a is an easy-to find constant in the coefficient field. For certain domains, such as the rational numbers, any irreducible multivariate polynomial $h(x, y)$ can be mapped to an irreducible univariate polynomial $h(x, b)$ for some constant b. This is the important *Hilbert irreducibility theorem*, whose consequence is that the combinatorial explosion observed in the univariate lifting algorithm is, in practice, unlikely. However, the magnitude and probabilistic distribution of good points b is not completely analyzed.

For so-called non-Hilbertian coefficient fields good reduction is not possible. An important such field is the complex number. Clearly, all $f(x, b)$ completely split into linear factors, while $f(x, y)$ may be irreducible over the complex numbers. An example of an irreducible polynomial is $f(x, y) = x^2 - y^3$. Polynomials that remain irreducible over the complex numbers are called absolutely irreducible. An additional problem is the determination of the algebraic extension of the ground field in which the absolutely irreducible factors can be expressed. In the example

$$x^6 - 2x^3y^2 + y^4 - 2x^3 = (x^3 - \sqrt{2}x - y^2) \cdot (x^3 + \sqrt{2}x - y^2)$$

the needed extension field is $\mathbb{Q}(\sqrt{2})$. The relation-finding approach proves successful for this problem. The root is computed as a Taylor series in y, and the integrality of the linear relation for the powers of the series means that the multipliers are polynomials in y of bounded degree. Several algorithms of polynomial-time complexity and pointers to the literature are found in Kaltofen [1995].

Bivariate polynomials constitute implicit representations of algebraic curves. It is an important operation in geometric modeling to convert from implicit to parametric representation. For example, the circle

$$x^2 + y^2 - 1 = 0$$

has the rational parameterization

$$x = \frac{2t}{1+t^2}, \qquad y = \frac{1-t^2}{1+t^2}, \quad \text{where } -\infty \le t \le \infty$$

Algorithms are known that can find such rational parameterizations provided that they exist [Sendra and Winkler 1991]. It is crucial that the inputs to these algorithms are absolutely irreducible polynomials.

8.4.4 Polynomials in Many Variables

Polynomials in many variables, such as the symmetric Toeplitz determinant previously exhibited, are rarely given explicitly, due to the fact that the number of possible terms grows exponentially in the number of variables: there can be as many as $\binom{n+v}{n} \ge 2^{\min\{n,v\}}$ terms in a polynomial of degree n with v variables. Even the factors may be dense in canonical representation, but could be sparse in another basis: for instance, the polynomial

$$(x_1 - 1)(x_2 - 2) \cdots (x_v - v) + 1$$

has only two terms in the shifted basis, whereas it has 2^v terms in the power basis, i.e., in expanded format.

Randomized algorithms are available that can efficiently compute a factor of an implicitly given polynomial, say, a matrix determinant, and even can find a shifted basis with respect to which a factor would be sparse, provided, of course, that such a shift exists. The approach is by manipulating polynomials in so-called black box representations [Kaltofen and Trager 1990]: a black box is an object that takes as input a value for each variable, and then produces the value of the polynomial it represents at the specified point. In the Toeplitz example the representation of the determinant could be the Gaussian elimination program which computes it. We note that the size of the polynomial in this case would be nearly constant, only the variable names and the dimension need to be stored. The factorization algorithm then outputs procedures which will evaluate all irreducible factors at an arbitrary point (supplied as the input). These procedures make calls to the black box given as input to the factorization algorithm in order to evaluate them at certain points, which are derived from the point at which the procedures computing the values of the factors are probed. It is, of course, assumed that subsequent calls evaluate one and the same factor and not associates that are scalar multiples of one another. The algorithm by Kaltofen and Trager [1990] finds procedures that with a controllably high probability evaluate the factors correctly. Randomization is needed to avoid parasitic factorizations of homomorphic images which provide some static data for the factor boxes and cannot be avoided without mathematical conjecture. The procedures that evaluate the individual factors are deterministic.

Factors constructed as black box programs are much more space efficient than those represented in other formats, for example, the straight-line program format [Kaltofen 1989]. More importantly, once the black box representation for the factors is found, sparse representations can be rapidly computed by any of the new sparse interpolation algorithms. See Grigoriev and Lakshman [1995] for the latest method allowing shifted bases and pointers to the literature of other methods, including those for the standard power bases.

The black box representation of polynomials is normally not supported by commercial computer algebra systems such as Axiom, Maple, or Mathematica. Díaz is currently developing the FoxBox system in C++ that makes black box methodology available to users of such systems. It is anticipated that factorizations as those of large symmetric Toeplitz determinants will be possible on computers. Earlier implementations based on the straight-line program model [Freeman et al. 1988] could factor 16 × 16 group determinants, which represent polynomials of over 300 million terms.

Acknowledgment

This material is based on work supported in part by the National Science Foundation under Grants CCR-9319776 (first and second author) and CCR-9020690 (third author), by GTE under a Graduate Computer Science Fellowship (first author), and by PSC CUNY Awards 665301 and 666327 (third author). Part of this work was done while the second author was at the Department of Computer Science at Rensselaer Polytechnic Insititute in Troy, New York.

Defining Terms

Characteristic polynomial: A polynomial associated with a square matrix, the determinant of the matrix when a single variable is subtracted to its diagonal entries. The roots of the characteristic polynomial are the eigenvalues of the matrix.

Condition number: A scalar derived from a matrix that measures its relative nearness to a singular matrix. Very close to singular means a large condition number, in which case numeric inversion becomes an unstable process.

Degree order: An order of the terms in a multivariate polynomial; for two variables x and y with $x \prec y$ the ascending chain of terms is $1 \prec x \prec y \prec x^2 \prec xy \prec y^2 \cdots$.

Determinant: A polynomial in the entries of a square matrix with the property that its value is nonzero if and only if the matrix is invertible.

Lexicographic order: An order of the terms in a multivariate polynomial; for two variables x and y with $x \prec y$ the ascending chain of terms is $1 \prec x \prec x^2 \prec \cdots \prec y \prec xy \prec x^2y \cdots \prec y^2 \prec xy^2 \cdots$.

Ops: Arithmetic operations, i.e., additions, subtractions, multiplications, or divisions; as in floating point operations (*flops*).

Singularity: A square matrix is singular if there is a nonzero second matrix such that the product of the two is the zero matrix. Singular matrices do not have inverses.

Sparse matrix: A matrix where many of the entries are zero.

Structured matrix: A matrix where each entry can be derived by a formula depending on few parameters. For instance, the Hilbert matrix has $1/(i + j - 1)$ as the entry in row i and column j.

References

Anderson, E. et al. 1992. *LAPACK Users' Guide*. SIAM Pub., Philadelphia, PA.

Aho, A., Hopcroft, J., and Ullman, J. 1974. *The Design and Analysis of Algorithms*. Addison–Wesley, Reading, MA.

Bareiss, E. H. 1968. Sylvester's identity and multistep integers preserving Gaussian elimination. *Math. Comp.* 22:565–578.

Becker, T. and Weispfenning, V. 1993. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer–Verlag, New York.

Berlekamp, E. R. 1967. Factoring polynomials over finite fields. *Bell Systems Tech. J.* 46:1853–1859; rev. 1968. *Algebraic Coding Theory*. Chap. 6, McGraw–Hill, New York.

Berlekamp, E. R. 1970. Factoring polynomials over large finite fields. *Math. Comp.* 24:713–735.

Bini, D. and Pan, V. Y. 1991. Parallel complexity of tridiagonal symmetric eigenvalue problem. In *Proc. 2nd Annu. ACM-SIAM Symp. on Discrete Algorithms*, pp. 384–393. ACM Press, New York, SIAM Pub., 1994. Philadelphia, PA.

Bini, D. and Pan, V. Y. 1994. *Polynomial and Matrix Computations Vol. 1, Fundamental Algorithms*. Birkhäuser, Boston, MA.

Bini, D. and Pan, V. Y. 1996. *Polynomial and Matrix Computations, Vol. 2*. Birkhäuser, Boston, MA.

Borodin, A. and Munro, I. 1975. *Computational Complexity of Algebraic and Numeric Problems*. American Elsevier, New York.

Brown, W. S. 1971. On Euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM* 18:478–504.

Brown, W. S. and Traub, J. F. 1971. On Euclid's algorithm and the theory of subresultants. *J. ACM* 18:505–514.

Buchberger, B. 1965. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* Ph.D. dissertation. University of Innsbruck, Austria.

Buchberger, B. 1976. A theoretical basis for the reduction of polynomials to canonical form. *ACM SIGSAM Bull.* 10(3):19–29.

Buchberger, B. 1983. A note on the complexity of constructing Gröbner-bases. In *Proc. EUROCAL '83*, J. A. van Hulzen, ed. *Lecture Notes in Computer Science*, pp. 137–145. Springer.

Buchberger, B. 1985. Gröbner bases: an algorithmic method in polynomial ideal theory. In *Recent Trends in Multidimensional Systems Theory*, N. K. Bose, ed., pp. 184–232. D. Reidel, Dordrecht, Holland.

Cantor, D. G. 1989. On arithmetical algorithms over finite fields. *J. Combinatorial Theory, Serol. A* 50:285–300.

Canny, J. 1990. Generalized characteristic polynomials. *J. Symbolic Comput.* 9(3):241–250.

Canny, J. and Emiris, I. 1993a. An efficient algorithm for the sparse mixed resultant. In *Proc. AAECC-10*, G. Cohen, T. Mora, and O. Moreno, ed. Vol. 673, *Lecture Notes in Computer Science*, pp. 89–104. Springer.

Canny, J. and Emiris, I. 1993b. A practical method for the sparse resultant. In *ISSAC '93*, *Proc. Internat. Symp. Symbolic Algebraic Comput.*, M. Bronstein, ed., pp. 183–192. ACM Press, New York.

Canny, J. and Emiris, I. 1994. Efficient incremental algorithms for the sparse resultant and the mixed volume. Tech. Rep., Univ. California-Berkeley, CA.

Canny, J., Kaltofen, E., and Lakshman, Y. 1989. Solving systems of non-linear polynomial equations faster. In *Proc. ACM-SIGSAM Internat. Symp. Symbolic Algebraic Comput.*, pp. 121–128.

Canny, J. and Manocha, D. 1991. Efficient techniques for multipolynomial resultant algorithms. In *ISSAC '91, Proc. Internat. Symp. Symbolic Algebraic Comput.*, S. M. Watt, ed., pp. 85–95, ACM Press, New York.

Cantor, D. G. and Zassenhaus, H. 1981. A new algorithm for factoring polynomials over finite fields. *Math. Comp.* 36:587–592.

Cayley, A. 1865. On the theory of eliminaton. *Cambridge and Dublin Math. J.* 3:210–270.

Chionh, E. 1990. *Base Points, Resultants and Implicit Representation of Rational Surfaces.* Ph.D. dissertation. Department of Computer Science, University of Waterloo, Waterloo, Canada.

Collins, G. E. 1967. Subresultants and reduced polynomial remainder sequences. *J. ACM* 14:128–142.

Collins, G. E. 1971. The calculation of multivariate polynomial resultants. *J. ACM* 18:515–532.

Cuppen, J. J. M. 1981. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.* 36:177–195.

Davenport, J. H., Tournier, E., and Siret, Y. 1988. *Computer Algebra Systems and Algorithms for Algebraic Computation*. Academic Press, London.

Díaz, A. and Kaltofen, E. 1995. On computing greatest common divisors with polynomials given by black boxes for their evaluation. In *ISSAC '95 Proc. 1995 Internat. Symp. Symbolic Algebraic Comput.*, A. H. M. Levelt, ed., pp. 232–239, ACM Press, New York.

Dixon, A. L. 1908. The elimination of three quantics in two independent variables. In *Proc. London Math. Soc.* Vol. 6, pp. 468–478.

Dongarra, J. et al. 1978. *LAPACK Users' Guide*. SIAM Pub., Philadelphia, PA.

Faugère, J. C., Gianni, P., Lazard, D., and Mora, T. 1993. Efficient computation of zero-dimensional Gröbner bases by change of ordering. *J. Symbolic Comput.* 16(4):329–344.

Ferguson, H. R. P. and Forcade, R. W. 1982. Multidimensional Euclidean algorithms. *J. Reine Angew. Math.* 334:171–181.

Finch, S. 1995. The miraculous Bailey–Borwein–Plouffe pi algorithm. Internet document, Mathsoft Inc., http://www.mathsoft.com/asolve/plouffe/plouffe.html, Oct.

Foster, L. V. 1981. Generalizations of Laguerre's method: higher order methods. *SIAM J. Numer. Anal.* 18:1004–1018.

Freeman, T. S., Imirzian, G., Kaltofen, E., and Lakshman, Y. 1988. Dagwood: a system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software* 14(3):218–240.

Garbow, B. S. et al. 1972. *Matrix Eigensystem Routines: EISPACK Guide Extension*. Springer, New York.

Geddes, K. O., Czapor, S. R., and Labahn, G. 1992. *Algorithms for Computer Algebra*. Kluwer Academic.

Gelfand, I. M., Kapranov, M. M., and Zelevinsky, A. V. 1994. *Discriminants, Resultants and Multidimensional Determinants*. Birkhäuser Verlag, Boston, MA.

George, A. and Liu, J. W.-H. 1981. *Computer Solution of Large Sparse Positive Definite Linear Systems*. Prentice–Hall, Englewood Cliffs, NJ.

Gianni, P. and Trager, B. 1985. GCD's and factoring polynomials using Gröbner bases. *Proc. EUROCAL '85*, Vol. 2, *Lecture Notes in Computer Science*, 204, pp. 409–410.

Giesbrecht, M. 1995. Nearly optimal algorithms for canonical matrix forms. *SIAM J. Comput.* 24(5):948–969.

Gilbert, J. R. and Tarjan, R. E. 1987. The analysis of a nested dissection algorithm. *Numer. Math.* 50:377–404.

Golub, G. H. and Van Loan, C. F. 1989. *Matrix Computations*. Johns Hopkins Univ. Press, Baltimore, MD.

Gondran, M. and Minoux, M. 1984. *Graphs and Algorithms*. Wiley–Interscience, New York.

Grigoriev, D. Y. and Lakshman, Y. N. 1995. Algorithms for computing sparse shifts for multivariate polynomials. In *ISSAC '95 Proc. 1995 Internat. Symp. Symbolic Algebraic Comput.*, A. H. M. Levelt, ed., pp. 96–103, ACM Press, New York.

Hansen, E., Patrick, M., and Rusnack, J. 1977. Some modifications of Laguerre's method. *BIT* 17:409–417.

Heath, M. T., Ng, E., and Peyton, B. W. 1991. Parallel algorithms for sparse linear systems. *SIAM Rev.* 33:420–460.

Jenkins, M. A., and Traub, J. F. 1970. A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration. *Numer. Math.* 14:252–263.

Kaltofen, E. 1982. Polynomial factorization. In 2nd ed. *Computer Algebra*, B. Buchberger, G. Collins, and R. Loos, eds., pp. 95–113. Springer–Verlag, Vienna.

Kaltofen, E. 1988. Greatest common divisors of polynomials given by straight-line programs. *J. ACM* 35(1):231–264.

Kaltofen, E. 1989. Factorization of polynomials given by straight-line programs. In *Randomness and Computation*, S. Micali, ed. Vol. 5 of Advances in computing research, pp. 375–412. JAI Press, Greenwhich, CT.

Kaltofen, E. 1990. Polynomial factorization 1982–1986. 1990. In *Computers in Mathematics*, D. V. Chudnovsky and R. D. Jenks, eds. Vol. 125, *Lecture Notes in Pure and Applied Mathematics,* pp. 285–309. Marcel Dekker, New York.

Kaltofen, E. 1992. Polynomial factorization 1987–1991. In *Proc. LATIN '92*, I. Simon, ed. Vol. 583, *Lecture Notes in Computer Science*, pp. 294–313.

Kaltofen, E. 1995. Effective Noether irreducibility forms and applications. *J. Comput. Syst. Sci.* 50(2):274–295.

Kaltofen, E., Krishnamoorthy, M. S., and Saunders, B. D. 1990. Parallel algorithms for matrix normal forms. *Linear Algebra Appl.* 136:189–208.

Kaltofen, E. and Lakshman, Y. 1988. Improved sparse multivariate polynomial interpolation algorithms. *Proc. ISSAC '88,* Vol. 358, *Lecture Notes in Computer Science,* pp. 467–474.

Kaltofen, E. and Lobo, A. 1994. Factoring high-degree polynomials by the black box Berlekamp algorithm. In *ISSAC '94, Proc. Internat. Symp. Symbolic Algebraic Comput.*, J. von zur Gathen and M. Giesbrecht, eds., pp. 90–98, ACM Press, New York.

Kaltofen, E., Musser, D. R., and Saunders, B. D. 1983. A generalized class of polynomials that are hard to factor. *SIAM J. Comp.* 12(3):473–485.

Kaltofen, E. and Pan, V. 1991. Processor efficient parallel solution of linear systems over an abstract field. In *Proc. 3rd Ann. ACM Symp. Parallel Algor. Architecture*, pp. 180–191, ACM Press, New York.

Kaltofen, E. and Pan, V. 1992. Processor-efficient parallel solution of linear systems II: the positive characteristic and singular cases. In *Proc. 33rd Annual Symp. Foundations of Comp. Sci.*, pp. 714–723, Los Alamitos, CA. IEEE Computer Society Press.

Kaltofen, E. and Shoup, V. 1995. Subquadratic-time factoring of polynomials over finite fields. In *Proc. 27th Annual ACM Symp. Theory Comp.*, pp. 398–406, ACM Press, New York.

Kaltofen, E. and Trager, B. 1990. Computing with polynomials given by black boxes for their evaluations: greatest common divisors, factorization, separation of numerators and denominators. *J. Symbolic Comput.* 9(3):301–320.

Kapur, D. 1986. Geometry theorem proving using Hilbert's nullstellensatz. *J. Symbolic Comp.* 2:399–408.

Kapur, D. and Lakshman, Y. N. 1992. Elimination methods: an introduction. In *Symbolic and Numerical Computation for Artificial Intelligence*. B. Donald, D. Kapur, and J. Mundy, eds. Academic Press.

Kapur, D. and Saxena, T. 1995. Comparison of various multivariate resultant formulations. In *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '95*, A. H. M. Levelt, ed., pp. 187–195, ACM Press, New York.

Kapur, D., Saxena, T., and Yang, L. 1994. Algebraic and geometric reasoning using Dixon resultants. In *ISSAC '94, Proc. Internat. Symp. Symbolic Algebraic Comput.* J. von zur Gathen and M. Giesbrecht, ed., pp. 99–107, ACM Press, New York.

Knuth, D. E. 1981. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms,* 2nd ed. Addison–Wesley, Reading, MA.

Lakshman, Y. N. 1990. *On the complexity of computing Gröbner bases for zero dimensional polynomia.* Ph.D. thesis, Dept. Comput. Sci., Rensselaer Polytechnic Inst. Troy, NY, Dec.

Lakshman, Y. N. and Saunders, B. D. 1994. On computing sparse shifts for univariate polynomials. In *ISSAC '94, Proc. Internat. Symp. Symbolic Algebraic Comput.*, J. von zur Gathen and M. Giesbrecht, eds., pp. 108–113, ACM Press, New York.

Lakshman, Y. N. and Saunders, B. D. 1995. Sparse polynomial interpolation in non-standard bases. *SIAM J. Comput.* 24(2):387–397.

Lazard, D. 1981. Résolution des systèmes d'équation algébriques. *Theoretical Comput. Sci.* 15:77–110. (In French).

Lenstra, A. K., Lenstra, H. W., and Lovász, L. 1982. Factoring polynomials with rational coefficients. *Math. Ann.* 261:515–534.

Leyland, P. 1995. Cunningham project data. Internet document, Oxford Univ., ftp://sable.ox.ac.uk/pub/math/cunningham/, November.

Lipton, R. J., Rose, D., and Tarjan, R. E. 1979. Generalized nested dissection. *SIAM J. on Numer. Analysis* 16(2):346–358.

Macaulay, F. S. 1916. Algebraic theory of modular systems. *Cambridge Tracts* 19, Cambridge.

MacWilliams, F. J. and Sloan, N. J. A. 1977. *The Theory of Error-Correcting Codes*. North–Holland, New York.

Madsen, K. 1973. A root-finding algorithm based on Newton's method. *BIT* 13:71–75.

McCormick, S., ed. 1987. *Multigrid Methods*. SIAM Pub., Philadelphia, PA.

McNamee, J. M. 1993. A bibliography on roots of polynomials. *J. Comput. Appl. Math.* 47(3):391–394.

Miller, V. 1992. Factoring polynomials via relation-finding. In *Proc. ISTCS '92*, D. Dolev, Z. Galil, and M. Rodeh, eds. Vol. 601, *Lecture Notes in Computer Science*, pp. 115–121.

Monagan, M. B. 1992. A heuristic irreducibility test for univariate polynomials. *J. Symbolic Comput.* 13(1):47–57.

Musser, D. R. 1975. Multivariate polynomial factorization. *J. ACM* 22:291–308.

Niederreiter, H. 1994. New deterministic factorization algorithms for polynomials over finite fields. In *Finite Fields: Theory, Applications and Algorithms*, L. Mullen and P. J.-S. Shiue, eds. Vol. 168, Contemporary mathematics, pp. 251–268, Amer. Math. Soc., Providence, RI.

Ortega, J. M., and Voight, R. G. 1985. Solution of partial differential equations on vector and parallel computers. *SIAM Rev.* 27(2):149–240.

Pan, V. Y. 1984a. How can we speed up matrix multiplication? *SIAM Rev.* 26(3):393–415.

Pan, V. Y. 1984b. How to multiply matrices faster. *Lecture Notes in Computer Science,* 179.

Pan, V. Y. 1987. Sequential and parallel complexity of approximate evaluation of polynomial zeros. *Comput. Math. (with Appls.)*, 14(8):591–622.

Pan, V. Y. 1991. Complexity of algorithms for linear systems of equations. In *Computer Algorithms for Solving Linear Algebraic Equations (State of the Art)*, E. Spedicato, ed. Vol. 77 of *NATO ASI Series*, Series F: computer and systems sciences, pp. 27–56, Springer–Verlag, Berlin.

Pan, V. Y. 1992a. Complexity of computations with matrices and polynomials. *SIAM Rev.* 34(2):225–262.

Pan, V. Y. 1992b. Linear systems of algebraic equations. In *Encyclopedia of Physical Sciences and Technology*, 2nd ed. Marvin Yelles, ed. Vol. 8, pp. 779–804, 1987. 1st ed. Vol. 7, pp. 304–329.

Pan, V. Y. 1993. Parallel solution of sparse linear and path systems. In *Synthesis of Parallel Algorithms*, J. H. Reif, ed. Ch. 14, pp. 621–678. Morgan Kaufmann, San Mateo, CA.

Pan, V. Y. 1994a. Improved parallel solution of a triangular linear system. *Comput. Math. (with Appl.)*, 27(11):41–43.

Pan, V. Y. 1994b. *On approximating polynomial zeros: modified quadtree construction and improved Newton's iteration.* Manuscript, Lehman College, CUNY, Bronx, New York.

Pan, V. Y. 1995a. Parallel computation of a Krylov matrix for a sparse and structured input. *Math. Comput. Modelling* 21(11):97–99.

Pan, V. Y. 1995b. *Solving a polynomial equation: some history and recent progress.* Manuscript, Lehman College, CUNY, Bronx, New York.

Pan, V. Y. 1996. Optimal and nearly optimal algorithms for approximating polynomial zeros. *Comput. Math. (with Appl.)*.

Pan, V. Y. and Preparata, F. P. 1995. Work-preserving speed-up of parallel matrix computations. *SIAM J. Comput.* 24(4):811–821.

Pan, V. Y. and Reif, J. H. 1992. Compact multigrid. *SIAM J. Sci. Stat. Comput.* 13(1):119–127.

Pan, V. Y. and Reif, J. H. 1993. Fast and efficient parallel solution of sparse linear systems. *SIAM J. Comp.*, 22(6):1227–1250.

Pan, V. Y., Sobze, I. and Atinkpahoun, A. 1995. On parallel computations with band matrices. *Inf. and Comput.* 120(2):227–250.

Parlett, B. 1980. *Symmetric Eigenvalue Problem*. Prentice–Hall, Englewood Cliffs, NJ.

Quinn, M. J. 1994. *Parallel Computing: Theory and Practice*. McGraw–Hill, New York.

Rabin, M. O. 1980. Probabilistic algorithms in finite fields. *SIAM J. Comp.* 9:273–280.

Renegar, J. 1989. On the worst case arithmetic complexity of approximating zeros of systems of polynomials. *SIAM J. Comput.* 18(2):350–370.

Ritt, J. F. 1950. *Differential Algebra*. AMS, New York.

Saad, Y. 1992. *Numerical Methods for Large Eigenvalue Problems: Theory and Algorithms*. Manchester Univ. Press, U.K., Wiley, New York. 1992.

Saad, Y. 1995. *Iterative Methods for Sparse Linear Systems*. PWS Kent, Boston, MA.

Sendra, J. R. and Winkler, F. 1991. Symbolic parameterization of curves. *J. Symbolic Comput.* 12(6): 607–631.

Shoup, V. 1996. A new polynomial factorization algorithm and its implementation. *J. Symbolic Comput.*

Smith, B. T. et al. 1970. *Matrix Eigensystem Routines: EISPACK Guide,* 2nd ed. Springer, New York.

St. Schwarz, 1956. On the reducibility of polynomials over a finite field. *Quart. J. Math. Oxford Ser. (2)*, 7:110–124.

Stederberg, T. and Goldman, R. 1986. Algebraic geometry for computer-aided design. *IEEE Comput. Graphics Appl.* 6(6):52–59.

Sturmfels, B. 1991. Sparse elimination theory. In *Proc. Computat. Algebraic Geom. and Commut. Algebra*, D. Eisenbud and L. Robbiano, eds. Cortona, Italy, June.

Sturmfels, B. and Zelevinsky, A. 1992. Multigraded resultants of the Sylvester type. *J. Algebra*.

von zur Gathen, J. and Shoup, V. 1992. Computing Frobenius maps and factoring polynomials. *Comput. Complexity* 2:187–224.

Walsh, P. G. 1993. *The computation of Puiseux expansions and a quantitative version of Runge's theorem on diophantine equations.* Ph.D. dissertation. University of Waterloo, Waterloo, Canada.

Watkins, D. S. 1982. Understanding the QR algorithm. *SIAM Rev.* 24:427–440.

Watkins, D. S. 1991. Some perspectives on the eigenvalue problem. *SIAM Rev.* 35(3):430–471.

Wilkinson, J. H. 1965. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, England.

Winkler, F. 1996. *Introduction to Computer Algebra*. Springer–Verlag, Heidelberg, Germany.
Wu, W. 1984. Basis principles of mechanical theorem proving in elementary geometries. *J. Syst. Sci. Math Sci.* 4(3):207–235.
Wu, W. 1986. Basis principles of mechanical theorem proving in elementary geometries. *J. Automated Reasoning* 2:219–252.
Zassenhaus, H. 1969. On Hensel factorization I. *J. Number Theory* 1:291–311.
Zippel, R. 1993. *Effective Polynomial Computations*, p. 384. Kluwer Academic, Boston, MA.

Further Information

The books by Knuth [1981], Davenport et al. [1988], Geddes et al. [1992], and Zippel [1993] provide a much broader introduction to the general subject. There are well-known libraries and packages of subroutines for the most popular numerical matrix computations, in particular, Dongarra et al. [1978] for solving linear systems of equations, Smith et al. [1970] and Garbow et al. [1972] approximating matrix eigenvalues, and Anderson et al. [1992] for both of the two latter computational problems. There is a comprehensive treatment of numerical matrix computations [Golub and Van Loan 1989], with extensive bibliography, and there are several more specialized books on them [George and Liu 1981, Wilkinson 1965, Parlett 1980, Saad 1992, 1995], as well as many survey articles [Heath et al. 1991, Watkins 1991, Ortega and Voight 1985, Pan 1992b] and thousands of research articles.

Special (more efficient) parallel algorithms have been devised for special classes of matrices, such as sparse [Pan and Reif 1993, Pan 1993], banded [Pan et al. 1995], and dense structured [Bini and Pan (cf. [1994])]. We also refer to Pan and Preparata [1995] on a simple but surprisingly effective extension of Brent's principle for improving the processor and work efficiency of parallel matrix algorithms and to Golub and Van Loan [1989], Ortega and Voight [1985], and Heath et al. [1991] on practical parallel algorithms for matrix computations.

9
Cryptography

Jonathan Katz
University of Maryland

9.1 Introduction

Cryptography is a vast subject, and we cannot hope to give a comprehensive account of the field here. Instead, we have chosen to narrow our focus to those areas of cryptography having the most practical relevance to the problem of *secure communication*. Broadly speaking, secure communication encompasses two complementary goals: the **secrecy** (sometimes called "privacy") and **integrity** of communicated data. These terms can be illustrated using the simple example of a user A sending a message m to a user B over a public channel. In the simplest sense, techniques for data secrecy ensure that an eavesdropping adversary (i.e., an adversary who sees all communication occurring on the channel) cannot get any information about m and, in particular, cannot determine m. Viewed in this way, such techniques protect against a *passive* adversary who listens to — but does not otherwise interfere with — the parties' communication. Techniques for data integrity, on the other hand, protect against an *active* adversary who may arbitrarily modify the data sent over the channel or may interject messages of his own. Here, secrecy is not necessarily an issue; instead, security in this setting requires only that any modifications performed by the adversary to the transmitted data will be detected by the receiving party.

In the cases of both secrecy and integrity, two different assumptions regarding the initial setup of the communicating parties can be considered. In the **private-key** setting (also known as the "shared-key," "secret-key," or "symmetric-key" setting), the assumption is that parties A and B have securely shared a random key s in advance. This key, which is completely hidden from the adversary, is used to secure their future communication. (We do not comment further on how such a key might be securely generated and shared; for our purposes, it is simply an assumption of the model.) Techniques for secrecy in this setting are called **private-key encryption** schemes, and those for data integrity are termed **message authentication codes (MACs)**.

In the **public-key** setting, the assumption is that one (or both) of the parties has generated a pair of keys: a *public key* that is widely disseminated throughout the network and an associated *secret key* that is kept private. The parties generating these keys may now use them to ensure secret communication using a **public-key encryption** scheme; they can also use these keys to provide data integrity (for messages they send) using a **digital signature scheme**.

We stress that, in the public-key setting, widespread distribution of the public key is assumed to occur before any communication over the public channel and without any interference from the adversary. In particular, if A generates a public/secret key, then B (for example) knows the correct public key and can use this key when communicating with A. On the flip side, the fact that the public key is widely disseminated implies that the adversary also knows the public key, and can attempt to use this knowledge when attacking the parties' communication.

We examine each of the above topics in turn. In Section 9.2 we introduce the information-theoretic approach to cryptography, describe some information-theoretic solutions for the above tasks, and discuss the severe limitations of this approach. We then describe the modern, computational (or complexity-theoretic) approach to cryptography that will be used in the remaining sections. This approach requires computational "hardness" assumptions of some sort; we formalize these assumptions in Section 9.3 and thus provide cryptographic building blocks for subsequent constructions. These building blocks are used to construct some basic cryptographic primitives in Section 9.4.

With these primitives in place, we proceed in the remainder of the chapter to give solutions for the tasks previously mentioned. Sections 9.5 and 9.6 discuss private-key encryption and message authentication, respectively, thereby completing our discussion of the private-key setting. Public-key encryption and digital signature schemes are described in Sections 9.7 and 9.8. We conclude with some suggestions for further reading.

9.2 Cryptographic Notions of Security

Two central features distinguish modern cryptography from "classical" (i.e., pre-1970s) cryptography: precise definitions and rigorous proofs of security. Without a precise definition of security for a stated goal, it is meaningless to call a particular protocol "secure." The importance of rigorous proofs of security (based on a set of well-defined assumptions) should also be clear: if a given protocol is not proven secure, there is always the risk that the protocol can be "broken." That protocol designers have not been able to find an attack does not preclude a more clever adversary from doing so. A proof that a given protocol is secure (with respect to some precise definition and using clearly stated assumptions) provides much more confidence in the protocol.

9.2.1 Information-Theoretic Notions of Security

With this in mind, we present one possible definition of security for private-key encryption and explore what can be achieved with respect to this definition. Recall the setting: two parties A and B share a random secret key s; this key will be used to secure their future communication and is completely hidden from the adversary. The data that A wants to communicate to B is called the **plaintext**, or simply the *message*. To transmit this message, A will **encrypt** the message using s and an encryption algorithm $\mathcal{E}$, resulting in **ciphertext** C. We write this as $C = \mathcal{E}_s(m)$. This ciphertext is sent over the public channel to B. Upon receiving the ciphertext, B recovers the original message by **decrypting** it using s and decryption algorithm $\mathcal{D}$; we write this as $m = \mathcal{D}_s(C)$.

We stress that the adversary is assumed to know the encryption and decryption algorithms; the only information hidden from the adversary is the secret key s. It is a mistake to require that the details of the encryption scheme be hidden in order for it to be secure, and modern cryptosystems are designed to be secure even when the full details of all algorithms are publicly available.

A plausible definition of security is to require that an adversary who sees ciphertext C (recall that C is sent over a public channel) — but does not know s — learns no information about the message m. In

particular, even if the message m is known to be one of two possible messages m_1, m_2 (each being chosen with probability 1/2), the adversary should not learn which of these two messages was actually sent. If we abstract this by requiring the adversary to, say, output "1" when he believes that m_1 was sent, this requirement can be formalized as:

> For all possible m_1, m_2 and for any adversary A, the probability that A guesses "1" when C is an encryption of m_1 is equal to the probability that A guesses "1" when C is an encryption of m_2.

That is, the adversary is no more likely to guess that m_1 was sent when m_1 is the actual message than when m_2 is the actual message. An encryption scheme satisfying this definition is said to be *information-theoretically secure* or to achieve *perfect secrecy*.

Perfect secrecy can be achieved by the **one-time pad** encryption scheme, which works as follows. Let ℓ be the length of the message m, where m is viewed as a binary string. The parties share in advance a secret key s that is uniformly distributed over strings of length ℓ (i.e., s is an ℓ-bit string chosen uniformly at random). To encrypt message m, the sender computes $C = m \oplus s$ where $\oplus$ represents binary exclusive-or and is computed bit-by-bit. Decryption is performed by setting $m = C \oplus s$. Clearly, decryption always recovers the original message. To see that the scheme is perfectly secret, let M, C, K be random variables denoting the message, ciphertext, and key, respectively, and note that for *any* message m and observed ciphertext c, we have:

$$\begin{aligned}\Pr[M = m|C = c] &= \frac{\Pr[C = c|M = m]\Pr[M = m]}{\Pr[C = c]} \\ &= \frac{\Pr[K = c \oplus m]\Pr[M = m]}{\Pr[C = c]} = \frac{2^{-\ell}\Pr[M = m]}{\Pr[C = c]}\end{aligned}$$

Thus, if m_1, m_2 have equal *a priori* probability, then $\Pr[M = m_1|C = c] = \Pr[M = m_2|C = c]$ and the ciphertext gives no further information about the actual message sent.

While this scheme is provably secure, it has limited value for most common applications. For one, *the length of the shared key is equal to the length of the message*. Thus, the scheme is simply impractical when long messages are sent. Second, it is easy to see that the scheme is secure *only when it is used to send a single message* (hence the name "one-time pad"). This will not do for applications is which multiple messages must be sent. Unfortunately, it can be shown that the one-time pad is optimal if perfect secrecy is desired. More formally, any scheme achieving perfect secrecy requires the key to be at least as long as the (total) length of all messages sent.

Can information-theoretic security be obtained for other cryptographic goals? It is known that perfectly-secure message authentication is possible (see, e.g., [51, Section 4.5]), although constructions achieving perfect security are similarly inefficient and require impractically long keys to authenticate multiple messages. In the public-key setting, the situation is even worse: perfectly secure public-key encryption or digital signature schemes are simply unachievable.

In summary, it is impossible to design perfectly secure yet practical protocols achieving the basic goals outlined in Section 9.1. To obtain reasonable solutions for our original goals, it will be necessary to (slightly) relax our definition of security.

9.2.2 Toward a Computational Notion of Security

The observation noted at the end of the previous section has motivated a shift in modern cryptography toward *computational* notions of security. Informally, whereas information-theoretic security guarantees that a scheme is absolutely secure against all (even arbitrarily powerful) adversaries, computational security ensures that a scheme is secure except with "negligible" probability against all "efficient" adversaries (we formally define these terms below). Although information-theoretic security is a strictly stronger notion, computational security suffices in practice and allows the possibility of more efficient schemes. However, it should be noted that computational security ultimately relies on currently unproven assumptions regarding the computational "hardness" of certain problems; that is, the security guarantee

provided in the computational setting is not as iron-clad as the guarantee given by information-theoretic security.

In moving to the computational setting, we introduce a *security parameter* $k \in \mathbb{N}$ that will be used to precisely define the terms "efficient" and "negligible." An *efficient* algorithm is defined as a probabilistic algorithm that runs in time polynomial in k; we also call such an algorithm "probabilistic, polynomial-time (PPT)." A *negligible* function is defined as one asymptotically smaller than any inverse polynomial; that is, a function $\varepsilon : \mathbb{N} \to \mathbb{R}^+$ is negligible if, for all $c \geq 0$ and for all n large enough, $\varepsilon(n) < 1/n^c$.

A cryptographic construction will be indexed by the security parameter k, where this value is given as input (in unary) to the relevant algorithms. Of course, we will require that these algorithms are all efficient and run in time polynomial in k. A typical definition of security in the computational setting requires that some condition hold for all PPT adversaries with all but negligible probability or, equivalently, that a PPT adversary will succeed in "breaking" the scheme with at most negligible probability. Note that the security parameter can be viewed as corresponding to a higher level of security (in some sense) because, as the security parameter increases, the adversary may run for a longer amount of time but has even lower probability of success.

Computational definitions of this sort will be used throughout the remainder of this chapter, and we explicitly contrast this type of definition with an information-theoretic one in Section 9.5 (for the case of private-key encryption).

9.2.3 Notation

Before continuing, we introduce some mathematical notation (following [30]) that will provide some useful shorthand. If A is a deterministic algorithm, then $y = A(x)$ means that we set y equal to the output of A on input x. If A is a probabilistic algorithm, the notation $y \leftarrow A(x_1, x_2, \ldots)$ denotes running A on inputs $x_1, x_2, \ldots$ and setting y equal to the output of A. Here, the "$\leftarrow$" is an explicit reminder that the process is probabilistic, and thus running A twice on the same inputs, for example, may not necessarily give the same value for y. If S represents a finite set, then $b \leftarrow S$ denotes assigning b an element chosen uniformly at random from S. If $p(x_1, x_2, \ldots)$ is a predicate that is either true or false, the notation

$$\Pr\left[x_1 \leftarrow S; x_2 \leftarrow A(x_1, y_2, \ldots); \cdots : p(x_1, x_2, \ldots)\right]$$

denotes the probability that $p(x_1, x_2, \ldots)$ is true after ordered execution of the listed experiment. The key features of this notation are that everything to the left of the colon represents the experiment itself (whose components are executed in order, from left to right, and are separated by semicolons) and the predicate is written to the right of the colon. To give a concrete example: $\Pr[b \leftarrow \{0, 1, 2\} : b = 2]$ denotes the probability that b is equal to 2 following the experiment in which b is chosen at random from $\{0, 1, 2\}$; this probability is, of course, $1/3$.

The notation $\{0, 1\}^\ell$ denotes the set of binary strings of length ℓ, while $\{0, 1\}^{\leq \ell}$ denotes the set of binary strings of length at most ℓ. We let $\{0, 1\}^*$ denote the set of finite-length binary strings. 1^k represents k repetitions of the digit "1", and has the value k in unary notation.

We assume familiarity with basic algebra and number theory on the level of [11]. We let $\mathbb{Z}_N = \{0, \ldots, N-1\}$ denote the set of integers modulo N; also, $\mathbb{Z}_N^* \subset \mathbb{Z}_N$ is the set of integers between 0 and N that are relatively prime to N. The Euler totient function is defined as $\varphi(N) \stackrel{\text{def}}{=} |\mathbb{Z}_N^*|$; of importance here is that $\varphi(p) = p - 1$ for p prime, and $\varphi(pq) = (p-1)(q-1)$ if p, q are distinct primes. For any N, the set $\mathbb{Z}_N^*$ forms a group under multiplication modulo N [11].

9.3 Building Blocks

As hinted at previously, cryptography seeks to exploit the presumed existence of computationally "hard" problems. Unfortunately, the mere existence of computationally hard problems does not appear to be sufficient for modern cryptography as we know it. Indeed, it is not currently known whether it is possible to have, say, secure private-key encryption (in the sense defined in Section 9.5) based only on the conjecture

that $P \neq NP$ (where P refers to those problems solvable in polynomial time and NP [informally] refers to those problems whose solutions can be verified in polynomial time; cf. [50] and Chapter 6). Seemingly stronger assumptions are currently necessary in order for cryptosystems to be built. On the other hand — fortunately for cryptographers — such assumptions currently seem very reasonable.

9.3.1 One-Way Functions

The most basic building block in cryptography is a **one-way function**. Informally, a one-way function f is a function that is "easy" to compute but "hard" to invert. Care must be taken, however, in interpreting this informal characterization. In particular, the formal definition of one-wayness requires that f be hard to invert *on average* and not merely hard to invert *in the worst case*. This is in direct contrast to the situation in complexity theory, where a problem falls in a particular class based on the worst-case complexity of solving it (and this is one reason why $P \neq NP$ does not seem to be sufficient for much of modern cryptography).

A number of equivalent definitions of one-way functions are possible; we present one such definition here. Note that the security parameter is explicitly given as input (in unary) to all algorithms.

Definition 9.1 Let $F = \{f_k : \mathcal{D}_k \to \mathcal{R}_k\}_{k \geq 1}$ be an infinite collection of functions where $\mathcal{D}_k \subseteq \{0,1\}^{\leq \ell(k)}$ for some fixed polynomial $\ell(\cdot)$. Then F is one-way (more formally, F is a one-way function family) if the following conditions hold:

"Easy" to compute There is a deterministic, polynomial-time algorithm A such that for all k and for all $x \in \mathcal{D}_k$ we have $A(1^k, x) = f_k(x)$.

"Hard" to invert For all PPT algorithms B, the following is negligible (in k):

$$\Pr[x \leftarrow \mathcal{D}_k; y = f_k(x); x' \leftarrow B(1^k, y) : f_k(x') = y].$$

Efficiently sampleable There is a PPT algorithm S such that $S(1^k)$ outputs a uniformly distributed element of $\mathcal{D}_k$.

It is not hard to see that the existence of a one-way function family implies $P \neq NP$. Thus, we have no hope of proving the unequivocal existence of a one-way function family given our current knowledge of complexity theory. Yet, certain number-theoretic problems appear to be one-way (and have thus far resisted all attempts at proving otherwise); we mention three popular candidates:

1. **Factoring.** Let $\mathcal{D}_k$ consist of pairs of k-bit primes, and define f_k such that $f_k(p, q) = pq$. Clearly, this function is easy to compute. It is also true that the domain $\mathcal{D}_k$ is efficiently sampleable because efficient algorithms for generating random primes are known (see, e.g., Appendix A.7 in [14]). Finally, f_k is hard to invert — and thus the above construction is a one-way function family — under the conjecture that factoring is hard (we refer to this simply as "the factoring assumption"). Of course, we have no proof for this conjecture; rather, evidence favoring the conjecture comes from the fact that no polynomial-time algorithm for factoring has been discovered in roughly 300 years of research related to this problem.
2. **Computing discrete logarithms.** Let $\mathcal{D}_k$ consist of tuples (p, g, x) in which p is a k-bit prime, g is a generator of the multiplicative group $\mathbb{Z}_p^*$, and $x \in \mathbb{Z}_{p-1}$. Furthermore, define f_k such that $f_k(p, g, x) = (p, g, g^x \bmod p)$. Given p, g as above and for any $y \in \mathbb{Z}_p^*$, define $\log_g y$ as the unique value $x \in \mathbb{Z}_{p-1}$ such that $g^x = y \bmod p$ (that a unique such x exists follows from the fact that $\mathbb{Z}_p^*$ is a cyclic group for p prime). Although exponentiation modulo p can be done in time polynomial in the lengths of p and the exponent x, it is not known how to efficiently compute $\log_g y$ given p, g, y. This suggests that this function family is indeed one-way (we note that there exist algorithms to efficiently sample from $\mathcal{D}_k$; see e.g., Chapter 6 in [14]).

   It should be clear that the above construction generalizes to other collections of finite, cyclic groups in which exponentiation can be done in polynomial time. Of course, the function family thus defined is one-way only if the discrete logarithm problem in the relevant group is hard. Other

popular examples in which this is believed to be the case include the group of points on certain elliptic curves (see Chapter 6 in [34]) and the subgroup of quadratic residues in $\mathbb{Z}_p^*$ when p and $\frac{p-1}{2}$ are both prime.

3. **RSA [45].** Let $\mathcal{D}_k$ consist of tuples (N, e, x), where N is a product of two distinct k-bit primes, $e < N$ is relatively prime to $\varphi(N)$, and $x \in \mathbb{Z}_N^*$. Furthermore, define f_k such that $f_k(N, e, x) = (N, e, x^e \bmod N)$. Following the previous examples, it should be clear that this function is easy to compute and has an efficiently sampleable domain (note that $\varphi(N)$ can be efficiently computed if p, q are known), It is conjectured that this function is hard to invert [45] and thus constitutes a one-way function family; we refer to this assumption simply as "the RSA assumption." For reasons of efficiency, the RSA function family is sometimes restricted by considering only $e = 3$ (and choosing N such that $\varphi(N)$ is not divisible by 3), and this is also believed to give a one-way function family.

   It is known that if RSA is a one-way function family, then factoring is hard (see the discussion of RSA as a trapdoor permutation, below). The converse is not believed to hold, and thus the RSA assumption appears to be strictly stronger than the factoring assumption (of course, all other things being equal, the weaker assumption is preferable).

9.3.2 Trapdoor Permutations

One-way functions are sufficient for many cryptographic applications. Sometimes, however, an "asymmetry" of sorts — whereby one party can efficiently accomplish some task which is infeasible for anyone else — must be introduced. **Trapdoor permutations** represent one way of formalizing this asymmetry. Recall that a one-way function has the property (informally) that it is "easy" to compute but "hard" to invert. Trapdoor permutations are also "easy" to compute and "hard" to invert *in general*; however, there is some trapdoor information that makes the permutation "easy" to invert. We give a formal definition now, and follow with some examples.

Definition 9.2 Let $\mathcal{K}$ be a PPT algorithm which, on input 1^k (for any $k \geq 1$), outputs a pair $(\mathsf{key}, \mathsf{td})$ such that key defines a permutation f_{key} over some domain $\mathcal{D}_{\mathsf{key}}$. We say $\mathcal{K}$ is a trapdoor permutation generator if the following conditions hold:

"Easy" to compute There is a deterministic, polynomial-time algorithm A such that for all k, all $(\mathsf{key}, \mathsf{td})$ output by $\mathcal{K}(1^k)$, and all $x \in \mathcal{D}_{\mathsf{key}}$ we have $A(1^k, \mathsf{key}, x) = f_{\mathsf{key}}(x)$.

"Hard" to invert For all PPT algorithms B, the following is negligible (in k):

$$\Pr[(\mathsf{key}, \mathsf{td}) \leftarrow \mathcal{K}(1^k); x \leftarrow \mathcal{D}_{\mathsf{key}}; y = f_{\mathsf{key}}(x); x' \leftarrow B(1^k, \mathsf{key}, y) : f_{\mathsf{key}}(x') = y].$$

Efficiently sampleable There is a PPT algorithm S such that for all $(\mathsf{key}, \mathsf{td})$ output by $\mathcal{K}(1^k)$, $S(1^k, \mathsf{key})$ outputs a uniformly distributed element of $\mathcal{D}_{\mathsf{key}}$.

"Easy" to invert with trapdoor There is a deterministic, polynomial-time algorithm I such that for all $(\mathsf{key}, \mathsf{td})$ output by $\mathcal{K}(1^k)$ and all $y \in \mathcal{D}_{\mathsf{key}}$ we have $I(1^k, \mathsf{td}, y) = f_{\mathsf{key}}^{-1}(y)$.

It should be clear that the existence of a trapdoor permutation generator immediately implies the existence of a one-way function family. Note that one could also define the completely analogous notion of trapdoor *function* generators; however, these have (thus far) had much more limited applications to cryptography.

It seems that the existence of a trapdoor permutation generator is a strictly stronger assumption than the existence of a one-way function family. Yet, number theory again provides examples of (conjectured) candidates:

9.3.2.1 RSA

We have seen in the previous section that RSA gives a one-way function family. It can also be used to give a trapdoor permutation generator. Here, we let $\mathcal{K}$ be an algorithm which, on input 1^k, chooses two distinct

k-bit primes p, q at random, sets $N = pq$, and chooses $e < N$ such that e and $\varphi(N)$ are relatively prime (note that $\varphi(N) = (p-1)(q-1)$ is efficiently computable because the factorization of N is known to $\mathcal{K}$). Then, $\mathcal{K}$ computes d such that $ed = 1 \bmod \varphi(N)$. The output is $((N, e), d)$, where (N, e) defines the permutation $f_{N,e} : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$ given by $f_{N,e}(x) = x^e \bmod N$. It is not hard to verify that this is indeed a permutation. That this permutation satisfies the first three requirements of the definition above follows from the fact that RSA is a one-way function family. To verify the last condition ("easiness" of inversion given the trapdoor d), note that

$$f_{N,d}(x^e \bmod N) = (x^e)^d \bmod N = x^{ed \bmod \varphi(N)} \bmod N = x,$$

and thus $f_{N,d} = f_{N,e}^{-1}$. So, the permutation $f_{N,e}$ can be efficiently inverted given d.

9.3.2.2 A Trapdoor Permutation Based on Factoring [42]

Let $\mathcal{K}$ be an algorithm which, on input 1^k, chooses two distinct k-bit primes p, q at random such that $p = q = 3 \bmod 4$, and sets $N = pq$. The output is $(N, (p, q))$, where N defines the permutation $f_N : \mathcal{QR}_N \to \mathcal{QR}_N$ given by $f_N(x) = x^2 \bmod N$; here, $\mathcal{QR}_N$ denotes the set of quadratic residues modulo N (i.e., the set of $x \in \mathbb{Z}_N^*$ such that x is a square modulo N). It can be shown that f_N is a permutation, and it is immediate that f_N is easy to compute. $\mathcal{QR}_N$ is also efficiently sampleable: to choose a random element in $\mathcal{QR}_N$, simply pick a random $x \in \mathbb{Z}_N^*$ and square it. It can also be shown that the trapdoor information p, q (i.e., the factorization of N) is sufficient to enable efficient inversion of f_N (see Section 3.6 in [14]). We now prove that this permutation is hard to invert as long as factoring is hard.

Lemma 9.1 *Assuming the hardness of factoring N of the form generated by $\mathcal{K}$, algorithm $\mathcal{K}$ described above is a trapdoor permutation family.*

Proof The lemma follows by showing that the squaring permutation described above is hard to invert (without the trapdoor). For any PPT algorithm B, define

$$\delta(k) \stackrel{\text{def}}{=} \Pr[(N, (p, q)) \leftarrow \mathcal{K}(1^k); y \leftarrow \mathcal{QR}_N; z \leftarrow B(1^k, N, y) : z^2 = y \bmod N]$$

(this is exactly the probability that B inverts a randomly-generated f_N). We use B to construct another PPT algorithm B' which factors the N output by $\mathcal{K}$. Algorithm B' operates as follows: on input $(1^k, N)$, it chooses a random $\tilde{x} \in \mathbb{Z}_N^*$ and sets $y = \tilde{x}^2 \bmod N$. It then runs $B(1^k, N, y)$ to obtain output z. If $z^2 = y \bmod N$ and $z \neq \pm\tilde{x}$, we claim that $\gcd(z - \tilde{x}, N)$ is a nontrivial factor of N. Indeed, $z^2 - \tilde{x}^2 = 0 \bmod N$, and thus

$$(z - \tilde{x})(z + \tilde{x}) = 0 \bmod N.$$

Since $z \neq \pm\tilde{x}$, it must be the case that $\gcd(z - \tilde{x}, N)$ gives a nontrivial factor of N, as claimed.

Now, conditioned on the fact that $z^2 = y \bmod N$ (which is true with probability $\delta(k)$), the probability that $z \neq \pm\tilde{x}$ is exactly 1/2; this follows from the fact that y has exactly four square roots, two of which are $\tilde{x}$ and $-\tilde{x}$. Thus, the probability that B' factors N is exactly $\delta(k)/2$. Because this quantity is negligible under the factoring assumption, $\delta(k)$ must be negligible as well. $\square$

9.4 Cryptographic Primitives

The building blocks of the previous section can be used to construct a variety of primitives, which in turn have a wide range of applications. We explore some of these primitives here.

9.4.1 Pseudorandom Generators

Informally, a **pseudorandom generator (PRG)** is a deterministic function that takes a short, random string as input and returns a longer, "random-looking" (i.e., pseudorandom) string as output. But to properly understand this, we must first ask: what does it mean for a string to "look random"? Of course, it is meaningless (in the present context) to talk about the "randomness" of any particular string — once a string is fixed, it is no longer random! Instead, we must talk about the randomness — or pseudorandomness — of a *distribution* of strings. Thus, to evaluate $G : \{0,1\}^k \to \{0,1\}^{k+1}$ as a PRG, we must compare the uniform distribution on strings of length $k+1$ with the distribution $\{G(x)\}$ for x chosen uniformly at random from $\{0,1\}^k$.

It is rather interesting that although the design and analysis of PRGs has a long history [33], it was not until the work of [9, 54] that a definition of PRGs appeared which was satisfactory for cryptographic applications. Prior to this work, the quality of a PRG was determined largely by ad hoc techniques; in particular, a PRG was deemed "good" if it passed a specific battery of statistical tests (for example, the probability of a "1" in the final bit of the output should be roughly 1/2). In contrast, the approach advocated by [9, 54] is that a PRG is good if it passes *all possible* (efficient) statistical tests! We give essentially this definition here.

Definition 9.3 Let $G : \{0,1\}^* \to \{0,1\}^*$ be an efficiently computable function for which $|G(x)| = \ell(|x|)$ for some fixed polynomial $\ell(k) > k$ (i.e., fixed-length inputs to G result in fixed-length outputs, and the output of G is always longer than its input). We say G is a pseudorandom generator (PRG) with expansion factor $\ell(k)$ if the following is negligible (in k) for all PPT statistical tests T:

$$\left|\Pr[x \leftarrow \{0,1\}^k : T(G(x)) = 1] - \Pr[y \leftarrow \{0,1\}^{\ell(k)} : T(y) = 1]\right|.$$

Namely, no PPT algorithm can distinguish between the output of G (on uniformly selected input) and the uniform distribution on strings of the appropriate length.

Given this strong definition, it is somewhat surprising that PRGs can be constructed at all; yet, they can be constructed from any one-way function (see below). As a step toward the construction of PRGs based on general assumptions, we first define and state the existence of a *hard-core bit* for any one-way function. Next, we show how this hard-core bit can be used to construct a PRG from any one-way *permutation*. (The construction of a PRG from arbitrary one-way *functions* is more complicated and is not given here.) This immediately extends to give explicit constructions of PRGs based on some specific assumptions.

Definition 9.4 Let $F = \{f_k : \mathcal{D}_k \to \mathcal{R}_k\}_{k\geq 1}$ be a one-way function family, and let $H = \{h_k : \mathcal{D}_k \to \{0,1\}\}_{k\geq 1}$ be an efficiently computable function family. We say that H is a hard-core bit for F if $h_k(x)$ is hard to predict with probability significantly better than 1/2 given $f_k(x)$. More formally, H is a hard-core bit for F the following is negligible (in k) for all PPT algorithms A:

$$\left|\Pr[x \leftarrow \mathcal{D}_k; y = f_k(x) : A(1^k, y) = h_k(x)] - 1/2\right|.$$

(Note that this is the "best" one could hope for in a definition of this sort, since an algorithm that simply outputs a random bit will guess $h_k(x)$ correctly half the time.)

We stress that *not* every H is a hard-core bit for a given one-way function family F. To give a trivial example: for the one-way function family based on factoring (in which $f_k(p,q) = pq$), it is easy to predict the last bit of p (and also q), which is always 1! On the other hand, a one-way function family with a hard-core bit can be constructed from any one-way function family; we state the following result to that effect without proof.

Theorem 9.2 ([27]) *If there exists a one-way function family F, then there exists (constructively) a one-way function family F' and an H which is a hard-core bit for F'.*

Hard-core bits for specific functions are known without recourse to the general theorem above [1, 9, 21, 32, 36]. We discuss a representative result for the case of RSA (this function family was introduced in Section 9.3, and we assume the reader is familiar with the notation used there). Let $H = \{h_k\}$ be a function family such that $h_k(N, e, x)$ returns the least significant bit of $x \bmod N$. Then H is a hard-core bit for RSA [1, 21]. Reiterating the definition above and assuming that RSA is a one-way function family, this means that given N, e, and $x^e \bmod N$ (for randomly chosen N, e, and x from the appropriate domains), it is hard for any PPT algorithm to compute the least significant bit of $x \bmod N$ with probability better than $1/2$.

We show now a construction of a PRG with expansion factor $k + 1$ based on any one-way *permutation* family $F = \{f_k\}$ with hard-core bit $H = \{h_k\}$. For simplicity, assume that the domain of f_k is $\{0, 1\}^k$; furthermore, for convenience, let $f(x), h(x)$ denote $f_{|x|}(x), h_{|x|}(x)$, respectively. Define:

$$G(x) = f(x) \circ h(x).$$

We claim that G is a PRG. As some intuition toward this claim, let $|x| = k$ and note that the first k bits of $G(x)$ are indeed uniformly distributed if x is uniformly distributed; this follows from the fact that f is a permutation over $\{0, 1\}^k$. Now, because H is a hard-core bit of F, $h(x)$ cannot be predicted by any efficient algorithm with probability better than $1/2$ even when the algorithm is given $f(x)$. Informally, then, $h(x)$ "looks random" to a PPT algorithm even conditioned on the observation of $f(x)$; hence, the entire string $f(x) \circ h(x)$ is pseudorandom.

It is known that given any PRG with expansion factor $k + 1$, it is possible to construct a PRG with expansion factor $\ell(k)$ for any polynomial $\ell(\cdot)$. The above construction, then, may be extended to yield a PRG that expands its input by an essentially arbitrary amount. Finally, although the preceding discussion focused only on the case of one-way *permutations*, it can be generalized (with much difficulty!) for the more general case of one-way *functions*. Putting these known results together, we obtain:

Theorem 9.3 ([31]) *If there exists a one-way function family, then for any polynomial $\ell(\cdot)$, there exists a PRG with stretching factor $\ell(k)$.*

9.4.2 Pseudorandom Functions and Block Ciphers

A pseudorandom generator G takes a short random string x and yields a polynomially-longer pseudorandom string $G(x)$. This in turn is useful in many contexts; see Section 9.5 for an example. However, a PRG has the following "limitations." First, for $G(x)$ to be pseudorandom, it is necessary that (1) x be chosen uniformly at random and also that (2) x be unknown to the distinguishing algorithm (clearly, once x is known, $G(x)$ is determined and hence no longer looks random). Furthermore, a PRG generates pseudorandom output whose length must be polynomially related to that of the input string x. For some applications, it would be nice to circumvent these limitations in some way.

These considerations have led to the definition and development of a more powerful primitive: a (family of) **pseudorandom functions (PRFs)**. Informally, a PRF $F : \{0, 1\}^k \times \{0, 1\}^m \to \{0, 1\}^n$ is a keyed function, so that fixing a particular key $s \in \{0, 1\}^k$ may be viewed as defining a function $F_s : \{0, 1\}^m \to \{0, 1\}^n$. (For simplicity in the rest of this and the following paragraph, we let $m = n = k$ although in general $m, n = \text{poly}(k)$.) Informally, a PRF F acts like a random function in the following sense: no efficient algorithm can distinguish the input/output behavior of F (with a randomly chosen key which is fixed for the duration of the experiment) from the input/output behavior of a truly random function. We stress that this holds even when the algorithm is allowed to interact with the function in an arbitrary way. It may be helpful to picture the following imaginary experiment: an algorithm is given access to a box that implements a function over $\{0, 1\}^k$. The algorithm can send inputs of its choice to the box and observe the corresponding outputs, but may not experiment with the box in any other way. Then F is a PRF if no efficient algorithm can distinguish whether the box implements a truly random function over $\{0, 1\}^k$ (i.e., a function chosen uniformly at random from the space of all 2^{k2^k} functions over $\{0, 1\}^k$) or whether it implements an instance of F_s (for uniformly chosen key $s \in \{0, 1\}^k$).

Note that this primitive is much stronger than a PRG. For one, the key s can be viewed as encoding an *exponential* amount of pseudorandomness because, roughly speaking, $F_s(x)$ is an independent pseudorandom value for each $x \in \{0,1\}^k$. Second, note that $F_s(x)$ is pseudorandom even if x is known, and even if x was not chosen at random. Of course, it must be the case that the key s is unknown and is chosen uniformly at random. We now give a formal definition of a PRF.

Definition 9.5 Let $\mathcal{F} = \{F_s : \{0,1\}^{m(k)} \to \{0,1\}^{n(k)}\}_{k \geq 1; s \in \{0,1\}^k}$ be an efficiently computable function family where $m, n = \mathsf{poly}(k)$, and let $\mathsf{Rand}_{m(k)}^{n(k)}$ denote the set of all functions from $\{0,1\}^{m(k)}$ to $\{0,1\}^{n(k)}$. We say $\mathcal{F}$ is a pseudorandom function family (PRF) if the following is negligible in k for all PPT algorithms A:

$$\left| \Pr[s \leftarrow \{0,1\}^k : A^{F_s(\cdot)}(1^k) = 1] - \Pr[f \leftarrow \mathsf{Rand}_{m(k)}^{n(k)} : A^{f(\cdot)}(1^k) = 1] \right|,$$

where the notation $A^{f(\cdot)}$ denotes that A has oracle access to function f; that is, A can send (as often as it likes) inputs of its choice to f and receive the corresponding outputs.

We do not present any details about the construction of a PRF based on general assumptions, beyond noting that they can be constructed from any one-way function family.

Theorem 9.4 ([25]) *If there exists a one-way function family F, then there exists (constructively) a PRF $\mathcal{F}$.*

An efficiently computable permutation family $\mathcal{P} = \{P_s : \{0,1\}^{m(k)} \to \{0,1\}^{m(k)}\}_{k \geq 1; s \in \{0,1\}^k}$ is an efficiently computable function family for which P_s is a permutation over $\{0,1\}^{m(k)}$ for each $s \in \{0,1\}^k$; and furthermore P_s^{-1} is efficiently computable (given s). By analogy with the case of a PRF, we say that $\mathcal{P}$ is a pseudorandom permutation (PRP) if P_s (with s randomly chosen in $\{0,1\}^k$) is indistinguishable from a truly random *permutation* over $\{0,1\}^{m(k)}$. A pseudorandom permutation can be constructed from any pseudorandom function [37].

What makes PRFs and PRPs especially useful in practice (especially as compared to PRGs) is that very efficient implementations of (conjectured) PRFs are available in the form of **block ciphers**. A block cipher is an efficiently computable permutation family $\mathcal{P} = \{P_s : \{0,1\}^m \to \{0,1\}^m\}_{s \in \{0,1\}^k}$ *for which keys have a fixed length k*. Because keys have a fixed length, we can no longer speak of a "negligible function" or a "polynomial-time algorithm" and consequently there is no notion of asymptotic security for block ciphers; instead, concrete security definitions are used. For example, a block cipher is said to be a (t, ε)-secure PRP, say, if no adversary running in time t can distinguish P_s (for randomly chosen s) from a random permutation over $\{0,1\}^m$ with probability better than ε. See [3] for further details.

Block ciphers are particularly efficient because they are *not* based on number-theoretic or algebraic one-way function families but are instead constructed directly, with efficiency in mind from the outset. One popular block cipher is DES (the Data Encryption Standard) [17, 38], which has 56-bit keys and is a permutation on $\{0,1\}^{64}$. DES dates to the mid-1970s, and recent concerns about its security — particularly its relatively short key length — have prompted the development* of a new block cipher termed AES (the Advanced Encryption Standard). This cipher supports 128-, 192-, and 256-bit keys, and is a permutation over $\{0,1\}^{128}$. Details of the AES cipher and the rationale for its construction are available [13].

9.4.3 Cryptographic Hash Functions

Although hash functions play an important role in cryptography, our discussion will be brief and informal because they are used sparingly in the remainder of this survey.

Hash functions — functions that compress long, often variable-length strings to much shorter strings — are widely used in many areas of computer science. For many applications, constructions of hash functions

*See http://csrc.nist.gov/CryptoToolkit/aes/ for a history and discussion of the design competition resulting in the selection of a cipher for AES.

with the necessary properties are known to exist without any computational assumptions. For cryptography, however, hash functions with very strong properties are often needed; furthermore, it can be shown that the existence of a hash function with these properties would imply the existence of a one-way function family (and therefore any such construction must be based on a computational assumption of some sort). We discuss one such property here.

The security property that arises most often in practice is that of collision resistance. Informally, H is said to be a **collision-resistant hash function** if an adversary is unable to find a "collision" in H; namely, two inputs x, x' with $x \neq x'$ but $H(x) = H(x')$. As in the case of PRFs and block ciphers (see the previous section), we can look at either the asymptotic security of a function family $\mathcal{H} = \{H_s : \{0,1\}^* \to \{0,1\}^k\}_{k\geq 1; s\in\{0,1\}^k}$ or the concrete security of a fixed hash function $H : \{0,1\}^* \to \{0,1\}^m$. The former are constructed based on specific computational assumptions, while the latter (as in the case of block ciphers) are constructed directly and are therefore much more efficient.

It is not hard to show that a collision-resistant hash function family mapping arbitrary-length inputs to fixed-length outputs is itself a one-way function family. Interestingly, however, collision-resistant hash function families are believed to be impossible to construct based on (general) one-way function families or trapdoor permutation generators [49]. On the other hand, constructions of collision-resistant hash function families based on specific computational assumptions (e.g., the hardness of factoring) are known; see Section 10.2 in [14].

In practice, customized hash functions — designed with efficiency in mind and not derived from number-theoretic problems — are used. One well-known example is MD5 [44], which hashes arbitrary-length inputs to 128-bit outputs. Because collisions in *any* hash function with output length k can be found in expected time (roughly) $2^{k/2}$ via a "birthday attack" (see, for example, Section 3.4.2 in [14]) and because computations on the order of 2^{64} are currently considered just barely outside the range of feasibility, hash functions with output lengths longer than 128 bits are frequently used. A popular example is SHA-1 [19], which hashes arbitrary-length inputs to 160-bit outputs. SHA-1 is considered collision-resistant for practical purposes, given current techniques and computational ability.

Hash functions used in cryptographic protocols sometimes require properties stronger than collision resistance in order for the resulting protocol to be provably secure [5]. It is fair to say that, in many cases, the exact properties needed by the hash function are not yet fully understood.

9.5 Private-Key Encryption

As discussed in Section 9.2.1, perfectly secret private-key encryption is achievable using the one-time pad encryption scheme; however, perfectly secret encryption requires that the shared key be at least as long as the communicated message. Our goal was to beat this bound by considering computational notions of security instead. We show here that this is indeed possible.

Let us first see what a definition of computational secrecy might involve. In the case of perfect secrecy, we required that for all messages m_0, m_1 of the same length ℓ, *no possible* algorithm could distinguish *at all* whether a given ciphertext is an encryption of m_0 or m_1. In the notation we have been using, this is equivalent to requiring that for all adversaries A,

$$\left|\Pr[s \leftarrow \{0,1\}^\ell : A(\mathcal{E}_s(m_0)) = 1] - \Pr[s \leftarrow \{0,1\}^\ell : A(\mathcal{E}_s(m_1)) = 1]\right| = 0.$$

To obtain a computational definition of security, we make two modifications: (1) we require the above to hold only for *efficient* (i.e., PPT) algorithms A; and (2) we only require the "distinguishing advantage" of the algorithm to be *negligible*, and not necessarily 0. The resulting definition of computational secrecy is that for all PPT adversaries A, the following is negligible:

$$\left|\Pr[s \leftarrow \{0,1\}^k : A(1^k, \mathcal{E}_s(m_0)) = 1] - \Pr[s \leftarrow \{0,1\}^k : A(1^k, \mathcal{E}_s(m_1)) = 1]\right|. \tag{9.1}$$

The one-time pad encryption scheme, together with the notion of a PRG as defined in Section 9.4.1, suggest a computationally secret encryption scheme in which the shared key is shorter than the message

(we reiterate that this is simply not possible if perfect secrecy is required). Specifically, let G be a PRG with expansion factor $\ell(k)$ (recall $\ell(k)$ is a polynomial with $\ell(k) > k$). To encrypt a message of length $\ell(k)$, the parties share a key s of length k; message m is then encrypted by computing $C = m \oplus G(s)$. Decryption is done by simply computing $m = C \oplus G(s)$.

For some intuition as to why this is secure, note that the scheme can be viewed as implementing a "pseudo"-one-time pad in which the parties share the pseudorandom string $G(s)$ instead of a uniformly random string of the same length. (Of course, to minimize the secret key length, the parties actually share s and regenerate $G(s)$ when needed.) But because the pseudorandom string $G(s)$ "looks random" to a PPT algorithm, the pseudo-one-time pad scheme "looks like" the one-time pad scheme to any PPT adversary. Because the one-time pad scheme is secure, so is the pseudo-one-time pad. (This is not meant to serve as a rigorous proof, but can easily be adapted to give one.)

We re-cap the discussion thus far in the following lemma.

Lemma 9.5 *Perfectly secret encryption is possible if and only if the shared key is at least as long as the message. However, if there exists a PRG, then there exists a computationally secret encryption scheme in which the message is (polynomially) longer than the shared key.*

Let us examine the pseudo-one-time pad encryption scheme a little more critically. Although the scheme allows encrypting messages longer than the secret key, the scheme is secure only when it is used *once* (as in the case of the one-time pad). Indeed, if an adversary views ciphertexts $C_1 = m_1 \oplus G(s)$ and $C_2 = m_2 \oplus G(s)$ (where m_1 and m_2 are unknown), the adversary can compute $m_1 \oplus m_2 = C_1 \oplus C_2$ and hence learn something about the relation between the two messages. Even worse, if the adversary somehow learns (or later determines), say, m_1, then the adversary can compute $G(s) = C_1 \oplus m_1$ and can thus decrypt any ciphertexts subsequently transmitted. We stress that such attacks (called *known-plaintext attacks*) are not merely of academic concern, because there are often messages sent whose values are uniquely determined, or known to lie in a small range. Can we obtain secure encryption even in the face of such attacks?

Before giving a scheme that prevents such attacks, let us precisely formulate a definition of security. First, the scheme should be "secure" even when used to encrypt multiple messages; in particular, an adversary who views the ciphertexts corresponding to multiple messages should not learn any information about the relationships among these messages. Second, the secrecy of the scheme should remain intact if some encrypted messages are known by the adversary. In fact, we can go beyond this last requirement and mandate that the scheme remain "secure" even if the adversary can request the encryption of messages of his choice (a *chosen-plaintext attack* of this sort arises when an adversary can influence the messages sent).

We model chosen-plaintext attacks by giving the adversary unlimited and unrestricted access to an *encryption oracle* denoted $\mathcal{E}_s(\cdot)$. This is simply a "black-box" that, on inputting a message m, returns an encryption of m using key s (in case $\mathcal{E}$ is randomized, the oracle chooses fresh randomness each time). Note that the resulting attack is perhaps stronger than what a real-world adversary can do (a real-world adversary likely cannot request as many encryptions — of arbitrary messages — as he likes); by the same token, if we can construct a scheme secure against this attack, then certainly the scheme will be secure in the real world. A formal definition of security follows.

Definition 9.6 A private-key encryption scheme $(\mathcal{E}, \mathcal{D})$ is said to be secure against chosen-plaintext attacks if, for all messages m_1, m_2 and all PPT adversaries A, the following is negligible:

$$\left|\Pr[s \leftarrow \{0,1\}^k : A^{\mathcal{E}_s(\cdot)}(1^k, \mathcal{E}_s(m_1)) = 1] - \Pr[s \leftarrow \{0,1\}^k : A^{\mathcal{E}_s(\cdot)}(1^k, \mathcal{E}_s(m_2)) = 1]\right|.$$

Namely, a PPT adversary cannot distinguish between the encryption of m_1 and m_2 *even if the adversary is given unlimited access to an encryption oracle.*

We stress one important corollary of the above definition: an encryption scheme secure against chosen-plaintext attacks *must be randomized* (in particular, the one-time pad does not satisfy the above definition).

This is so for the following reason: if the scheme were deterministic, an adversary could obtain $C_1 = \mathcal{E}_s(m_1)$ and $C_2 = \mathcal{E}_s(m_2)$ from its encryption oracle and then compare the given ciphertext to each of these values; thus, the adversary could immediately tell which message was encrypted. Our strong definition of security forces us to consider more complex encryption schemes.

Fortunately, many encryption schemes satisfying the above definition are known. We present two examples here; the first is mainly of theoretical interest (but is also practical for short messages), and its simplicity is illuminating. The second is more frequently used in practice.

Our first encryption scheme uses a key of length k to encrypt messages of length k (we remind the reader, however, that this scheme will be a tremendous improvement over the one-time pad because the present scheme can be used to encrypt polynomially-many messages). Let $\mathcal{F} = \{F_s : \{0,1\}^k \to \{0,1\}^k\}_{k \geq 1; s \in \{0,1\}^k}$ be a PRF (cf. Section 9.4.2); alternatively, one can think of k as being fixed and using a block cipher for $\mathcal{F}$ instead. We define encryption using key s as follows [26]: on input a message $m \in \{0,1\}^k$, choose a random $r \in \{0,1\}^k$ and output $\langle r, F_s(r) \oplus m \rangle$. To decrypt ciphertext $\langle r, c \rangle$ using key s, simply compute $m = c \oplus F_s(r)$.

We give some intuition for the security of this scheme against chosen-plaintext attacks. Assume the adversary queries the encryption oracle n times, receiving in return the ciphertexts $\langle r_1, c_1 \rangle, \ldots, \langle r_n, c_n \rangle$ (the messages to which these ciphertexts correspond are unimportant). Let the ciphertext given to the adversary — corresponding to the encryption of either m_1 or m_2 — be $\langle r, c \rangle$. By the definition of a PRF, the value $F_s(r)$ "looks random" to the PPT adversary A unless $F_s(\cdot)$ was previously computed on input r; in other words, $F_s(r)$ "looks random" to A unless $r \in \{r_1, \ldots, r_n\}$ (we call this occurrence a *collision*). Security of the scheme is now evident from the following: (1) assuming a collision does *not* occur, $F_s(r)$ is pseudorandom as discussed and hence the adversary cannot determine whether m_1 or m_2 was encrypted (as in the one-time pad scheme); furthermore, (2) the probability that a collision occurs is $\frac{n}{2^k}$, which is negligible (because n is polynomial in k). We thus have Theorem 9.6.

Theorem 9.6 ([26]) *If there exists a PRF $\mathcal{F}$, then there exists an encryption scheme secure against chosen-plaintext attacks.*

The previous construction applies to small messages whose length is equal to the output length of the PRF. From a theoretical point of view, an encryption scheme (secure against chosen-plaintext attacks) for longer messages follows immediately from the construction given previously; namely, to encrypt message $M = m_1, \ldots, m_\ell$ (where $m_i \in \{0,1\}^k$), simply encrypt each block of the message using the previous scheme, giving ciphertext:

$$\langle r_1, F_s(r_1) \oplus m_1, \ldots, r_\ell, F_s(r_\ell) \oplus m_\ell \rangle.$$

This approach gives a ciphertext twice as long as the original message and is therefore not very practical.

A better idea is to use a **mode of encryption**, which is a method for encrypting long messages using a block cipher with fixed input/output length. Four modes of encryption were introduced along with DES [18], and we discuss one such mode here (not all of the DES modes of encryption are secure). In cipher block chaining (CBC) mode, a message $M = m_1, \ldots, m_\ell$ is encrypted using key s as follows:

$$\begin{aligned}
&\text{Choose } C_0 \in \{0,1\}^k \text{ at random}\\
&\text{For } i = 1 \text{ to } \ell\text{:}\\
&\quad C_i = F_s(m_i \oplus C_{i-1})\\
&\text{Output } \langle C_0, C_1, \ldots, C_\ell \rangle
\end{aligned}$$

Decryption of a ciphertext $\langle C_0, \ldots, C_\ell \rangle$ is done by reversing the above steps:

$$\begin{aligned}
&\text{For } i = 1 \text{ to } \ell\text{:}\\
&\quad m_i = F_s^{-1}(C_i) \oplus C_{i-1}\\
&\text{Output } m_1, \ldots, m_\ell
\end{aligned}$$

It is known that CBC mode is secure against chosen-plaintext attacks [3].

9.6 Message Authentication

The preceding section discussed how to achieve message *secrecy*; we now discuss techniques for message *integrity*. In the private-key setting, this is accomplished using message authentication codes (MACs). We stress that secrecy and authenticity are two incomparable goals, and it is certainly possible to achieve either one without the other. As an example, the one-time pad — which achieves perfect secrecy — provides no message integrity whatsoever because *any* ciphertext C of the appropriate length decrypts to some valid message. Even worse, if C represents the encryption of a particular message m (so that $C = m \oplus s$ where s is the shared key), then flipping the first bit of C has the effect of flipping the first bit of the resulting decrypted message.

Before continuing, let us first define the semantics of a MAC.

Definition 9.7 A message authentication code consists of a pair of PPT algorithms $(\mathcal{T}, \mathsf{Vrfy})$ such that (here, the length of the key is taken to be the security parameter):

- The tagging algorithm $\mathcal{T}$ takes as input a key s and a message m and outputs a tag $t = \mathcal{T}_s(m)$.
- The verification algorithm Vrfy takes as input a key s, a message m, and a (purported) tag t and outputs a bit signifying acceptance (1) or rejection (0).

We require that for all m and all t output by $\mathcal{T}_s(m)$ we have $\mathsf{Vrfy}_s(m, t) = 1$.

Actually, a MAC should also be defined over a particular message space and this must either be specified or else clear from the context.

Schemes designed to detect "random" modifications of a message (e.g., error-correcting codes) do not constitute secure MACs because they are not designed to provide message authenticity in an *adversarial* setting. Thus, it is worth considering carefully the exact security goal we desire. Ideally, even if an adversary can request tags for multiple messages $m_1, \ldots$ of his choice, it should be impossible for the adversary to "forge" a valid-looking tag t on a new message m. (As in the case of encryption, this adversary is likely stronger than what is encountered in practice; however, if we can achieve security against even this strong attack so much the better!) To formally model this, we give the adversary access to an oracle $\mathcal{T}_s(\cdot)$, which returns a tag t for any message m of the adversary's choice. Let $m_1, \ldots, m_\ell$ denote the messages submitted by the adversary to this oracle. We say a forgery occurs if the adversary outputs (m, t) such that $m \notin \{m_1, \ldots, m_\ell\}$ and $\mathsf{Vrfy}_s(m, t) = 1$. Finally, we say a MAC is secure if the probability of a forgery is negligible for all PPT adversaries A. For completeness, we give a formal definition following [4].

Definition 9.8 MAC $(\mathcal{T}, \mathsf{Vrfy})$ is said to be secure against adaptive chosen-message attacks if, for all PPT adversaries A, the following is negligible:

$$\Pr[s \leftarrow \{0,1\}^k; (m, t) \leftarrow A^{\mathcal{T}_s(\cdot)}(1^k) : \mathsf{Vrfy}_s(m, t) = 1 \wedge m \notin \{m_1, \ldots, m_\ell\}],$$

where $m_1, \ldots, m_\ell$ are the messages that A submitted to $\mathcal{T}_s(\cdot)$.

We now give two constructions of a secure MAC. For the first, let $\mathcal{F} = \{F_s : \{0,1\}^k \to \{0,1\}^k\}_{k \geq 1; s \in \{0,1\}^k}$ be a PRF (we can also let $\mathcal{F}$ be a block cipher for some fixed value k). The discussion of PRFs in Section 9.4.2 should motivate the following construction of a MAC for messages of length k [26]: the tagging algorithm $\mathcal{T}_s(m)$ (where $|s| = |m| = k$) returns $t = F_s(m)$, and the verification algorithm $\mathsf{Vrfy}_s(m, t)$ outputs 1 if and only if $F_s(m) = t$. A proof of security for this construction is immediate: Let $m_1, \ldots, m_\ell$ denote those messages for which adversary A has requested a tag from $\mathcal{T}_s(\cdot)$. Because $\mathcal{F}$ is a PRF, $\mathcal{T}_s(m) = F_s(m)$ "looks random" for any $m \notin \{m_1, \ldots, m_\ell\}$ (call m of this sort *new*). Thus, the adversary's probability of outputting (m, t) such that $t = F_s(m)$ and m is new is (roughly) 2^{-k}; that is, the probability of guessing the output of a random function with output length k at a particular point m. This is negligible, as desired.

Because PRFs exist for any (polynomial-size) input length, the above construction can be extended to achieve secure message authentication for polynomially-long messages. We summarize the theoretical implications of this result in Theorem 9.7.

Theorem 9.7 ([26]) *If there exists a PRF $\mathcal{F}$, then there exists a MAC secure against adaptive chosen-message attack.*

Although the above result gives a theoretical solution to the problem of message authentication (and can be made practical for short messages by using a block cipher to instantiate the PRF), it does not give a practical solution for authenticating long messages. So, we conclude this section by showing a practical and widely used MAC construction for long messages. Let $\mathcal{F} = \{F_s : \{0,1\}^n \rightarrow \{0,1\}^n\}_{s \in \{0,1\}^k}$ denote a block cipher. For fixed ℓ, define the CBC-MAC for messages of length $(\{0,1\}^n)^\ell$ as follows (note the similarity with the CBC mode of encryption from Section 9.5): the tag of a message $m_1, \ldots, m_\ell$ with $m_i \in \{0,1\}^n$ is computed as:

$$\begin{aligned}
&C_0 = 0^n \\
&\text{For } i = 1 \text{ to } \ell: \\
&\quad C_i = F_s(m_i \oplus C_{i-1}) \\
&\text{Output } C_\ell
\end{aligned}$$

Verification of a tag t on a message $m_1, \ldots, m_\ell$ is done by re-computing C_ℓ as above and outputting 1 if and only if $t = C_\ell$. It is known that the CBC-MAC is secure against adaptive chosen-message attacks [4] for n sufficiently large. We stress that this is true only when fixed-length messages are authenticated (this was why ℓ was fixed, above). Subsequent work has focused on extending CBC-MAC to allow authentication of arbitrary-length messages [8, 41].

9.7 Public-Key Encryption

The advent of public-key encryption [15, 39, 45] marked a revolution in the field of cryptography. For hundreds of years, cryptographers had relied exclusively on shared, secret keys to achieve secure communication. Public-key cryptography, however, enables two parties to secretly communicate without having arranged for any *a priori* shared information. We first describe the semantics of a public-key encryption scheme, and then discuss two general ways such a scheme can be used.

Definition 9.9 A public-key encryption scheme is a triple of PPT algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ such that:

- The key generation algorithm $\mathcal{K}$ takes as input a security parameter 1^k and outputs a public key PK and a secret key SK.
- The encryption algorithm $\mathcal{E}$ takes as input a public key PK and a message m and outputs a ciphertext C. We write this as $C \leftarrow \mathcal{E}_{PK}(m)$.
- The deterministic decryption algorithm $\mathcal{D}$ takes as input a secret key SK and a ciphertext C and outputs a message m. We write this as $m = \mathcal{D}_{SK}(C)$.

We require that for all k, all (PK, SK) output by $\mathcal{K}(1^k)$, for all m, and for all C output by $\mathcal{E}_{PK}(m)$, we have $\mathcal{D}_{SK}(C) = m$.

For completeness, a message space must be specified; however, the message space is generally taken to be $\{0,1\}^*$.

There are a number of ways in which a public-key encryption scheme can be used to enable communication between a sender $\mathcal{S}$ and a receiver $\mathcal{R}$. First, we can imagine that when $\mathcal{S}$ and $\mathcal{R}$ wish to communicate, $\mathcal{R}$ executes algorithm $\mathcal{K}$ to generate the pair of keys (PK, SK). The public key PK is sent (in the clear) to $\mathcal{S}$, and the secret key SK is (of course) kept secret by $\mathcal{R}$. To send a message m, $\mathcal{S}$ computes $C \leftarrow \mathcal{E}_{PK}(m)$ and transmits C to $\mathcal{R}$. The receiver $\mathcal{R}$ can now recover the original message by computing $m = \mathcal{D}_{SK}(C)$. Note that to fully ensure secrecy against an eavesdropping adversary, it must be the case that m remains hidden even if the adversary sees both PK and C (i.e., the adversary eavesdrops on the *entire* communication between $\mathcal{S}$ and $\mathcal{R}$).

A second way to picture the situation is to imagine that $\mathcal{R}$ runs $\mathcal{K}$ to generate keys (PK, SK) *independent of any particular sender* $\mathcal{S}$ (indeed, the identity of $\mathcal{S}$ need not be known at the time the keys are generated). The public key PK of $\mathcal{R}$ is then widely distributed — for example, published on $\mathcal{R}$'s personal homepage — and may be used by *anyone* wishing to securely communicate with $\mathcal{R}$. Thus, when a sender $\mathcal{S}$ wishes to confidentially send a message m to $\mathcal{R}$, the sender simply looks up $\mathcal{R}$'s public key PK, computes $C \leftarrow \mathcal{E}_{PK}(m)$, and sends C to $\mathcal{R}$; decryption by $\mathcal{R}$ is done as before. In this way, multiple senders can communicate multiple times with $\mathcal{R}$ using the same public key PK for all communication.

Note that, as was the case above, secrecy must be guaranteed even when an adversary knows PK. This is so because, by necessity, $\mathcal{R}$'s public key is widely distributed so that anyone can communicate with $\mathcal{R}$. Thus, it is only natural to assume that the adversary also knows PK. The following definition of security extends the definition given in the case of private-key encryption.

Definition 9.10 A public-key encryption scheme $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ is said to be secure against chosen-plaintext attacks if, for all messages m_1, m_2 and all PPT adversaries A, the following is negligible:

$$\left|\Pr[(PK, SK) \leftarrow \mathcal{K}(1^k) : A(PK, \mathcal{E}_{PK}(m_0)) = 1] - \Pr[(PK, SK) \leftarrow \mathcal{K}(1^k) : A(PK, \mathcal{E}_{PK}(m_1) = 1]\right|.$$

The astute reader will notice that this definition is analogous to the definition of one-time security for private-key encryption (with the exception that the adversary is now given the public key as input), but seems inherently different from the definition of security against chosen-plaintext attacks (cf. Definition 9.6). Indeed, the above definition makes no mention of any "encryption oracle" as does Definition 9.6. However, it is known for the case of public-key encryption that the definition above implies security against chosen-plaintext attacks (of course, we have seen already that the definitions are *not* equivalent in the private-key setting).

Definition 9.10 has the following immediate and important consequence, first noted by Goldwasser and Micali [29]: for a public-key encryption scheme to be secure, encryption *must* be probabilistic. To see this, note that if encryption were deterministic, an adversary could always tell whether a given ciphertext C corresponds to an encryption of m_1 or m_2 by simply computing $\mathcal{E}_{PK}(m_1)$ and $\mathcal{E}_{PK}(m_2)$ himself (recall the adversary knows PK) and comparing the results to C.

The definition of public-key encryption — in which determining the message corresponding to a ciphertext is "hard" in general, but becomes "easy" with the secret key — is reminiscent of the definition of trapdoor permutations. Indeed, the following feasibility result is known.

Theorem 9.8 ([54]) *If there exists a trapdoor permutation (generator), there exists a public-key encryption scheme secure against chosen-plaintext attacks.*

Unfortunately, public-key encryption schemes constructed via this generic result are generally quite inefficient, and it is difficult to construct *practical* encryption schemes secure in the sense of Definition 9.10. At this point, some remarks about the practical efficiency of public-key encryption are in order. Currently known public-key encryption schemes are roughly three orders of magnitude slower (per bit of plaintext) than private-key encryption schemes with comparable security. For encrypting long messages, however, all is not lost: in practice, a long message m is encrypted by first choosing at random a "short" (i.e., 128-bit) key s, encrypting this key using a public-key encryption scheme, and then encrypting m using a private-key scheme with key s. So, the public-key encryption of m under public key PK is given by:

$$\langle \mathcal{E}_{PK}(s) \circ \mathcal{E}'_s(m) \rangle,$$

where $\mathcal{E}$ is the public-key encryption algorithm and $\mathcal{E}'$ represents a private-key encryption algorithm. If both the public-key and private-key components are secure against chosen-plaintext attacks, so is the scheme above. Thus, the problem of designing efficient public-key encryption schemes for long messages is reduced to the problem of designing efficient public-key encryption for short messages.

We discuss the well-known El Gamal encryption scheme [16] here. Let G be a cyclic (multiplicative) group of order q with generator $g \in G$. Key generation consists of choosing a random $x \in \mathbb{Z}_q$ and setting $y = g^x$. The public key is (G, q, g, y) and the secret key is x. To encrypt a message $m \in G$, the sender chooses a random $r \in \mathbb{Z}_q$ and sends:

$$\langle g^r, y^r m \rangle.$$

To decrypt a ciphertext $\langle A, B \rangle$ using secret key x, the receiver computes $m = B/A^x$. It is easy to see that decryption correctly recovers the intended message.

Clearly, security of the scheme requires the discrete logarithm problem in G to be hard; if the discrete logarithm problem were easy, then the secret key x could be recovered from the information contained in the public key. Hardness of the discrete logarithm problem is not, however, sufficient for the scheme to be secure in the sense of Definition 9.10; a stronger assumption (first introduced by Diffie and Hellman [15] and hence called the *decisional Diffie-Hellman (DDH) assumption*) is, in fact, needed. (See [52] or [7] for further details.)

We have thus far *not* mentioned the "textbook RSA" encryption scheme. Here, key generation results in public key (N, e) and secret key d such that $ed = 1 \bmod \varphi(N)$ (see Section 9.3.2 for further details) and encryption of message $m \in \mathbb{Z}_N^*$ is done by computing $C = m^e \bmod N$. The reason for its omission is that this scheme is simply *not secure* in the sense of Definition 9.10; for one thing, encryption in this scheme is deterministic and therefore cannot possibly be secure.

Of course — and as discussed in Section 9.3.2 — the RSA assumption gives a trapdoor permutation generator, which in turn can be used to construct a secure encryption scheme (cf. Theorem 9.8). Such an approach, however, is inefficient and not used in practice. The public-key encryption schemes used in practice that are based on the RSA problem seem to require additional assumptions regarding certain hash functions; we refer to [5] for details that are beyond our present scope.

We close this section by noting that current, widely used encryption schemes in fact satisfy stronger definitions of security than that of Definition 9.10; in particular, encryption schemes are typically designed to be secure against chosen-ciphertext attacks (see [7] for a definition). Two efficient examples of encryption schemes meeting this stronger notion of security include the Cramer-Shoup encryption scheme [12] (based on the DDH assumption) and OAEP-RSA [6, 10, 22, 48] (based on the RSA assumption and an assumption regarding certain hash functions [5]).

9.8 Digital Signature Schemes

As public-key encryption is to private-key encryption, so are digital signature schemes to message authentication codes. Digital signature schemes are the public-key analog of MACs; they allow a *signer* who has established a public key to "sign" messages in a way that is verifiable to anyone who knows the signer's public key. Furthermore (by analogy with MACs), no adversary can forge valid-looking signatures on messages that were not explicitly authenticated (i.e., signed) by the legitimate signer.

In more detail, to use a signature scheme, a user first runs a key generation algorithm to generate a public-key/private-key pair (PK, SK); the user then publishes and widely distributes PK (as in the case of public-key encryption). When the user wants to authenticate a message m, she may do so using the signing algorithm along with her secret key SK; this results in a signature σ. Now, *anyone* who knows PK can verify correctness of the signature by running the public verification algorithm using the known public key PK, message m, and (purported) signature σ. We formalize the semantics of digital signature schemes in the following definition.

Definition 9.11 A signature scheme consists of a triple of PPT algorithms $(\mathcal{K}, \mathsf{Sign}, \mathsf{Vrfy})$ such that:

- The key generation algorithm $\mathcal{K}$ takes as input a security parameter 1^k and outputs a public key PK and a secret key SK.

- The signing algorithm Sign takes as input a secret key SK and a message m and outputs a signature $\sigma = \mathsf{Sign}_{SK}(m)$.
- The verification algorithm Vrfy takes as input a public key PK, a message m, and a (purported) signature σ and outputs a bit signifying acceptance (1) or rejection (0).

We require that for all (PK, SK) output by $\mathcal{K}$, for all m, and for all σ output by $\mathsf{Sign}_{SK}(m)$, we have $\mathsf{Vrfy}_{PK}(m, \sigma) = 1$.

As in the case of MACs, the message space for a signature scheme should be specified. This is also crucial when discussing the security of a scheme.

A definition of security for signature schemes is obtainable by a suitable modification of the definition of security for MACs* (cf. Definition 9.8) with oracle $\mathsf{Sign}_{SK}(\cdot)$ replacing oracle $\mathcal{T}_s(\cdot)$, and the adversary now having as additional input the signer's public key. For reference, the definition (originating in [30]) is included here.

Definition 9.12 Signature scheme $(\mathcal{K}, \mathsf{Sign}, \mathsf{Vrfy})$ is said to be secure against adaptive chosen-message attacks if, for all PPT adversaries A, the following is negligible:

$$\Pr\left[(PK, SK) \leftarrow \mathcal{K}(1^k); (m, \sigma) \leftarrow A^{\mathsf{Sign}_{SK}(\cdot)}(1^k, PK) : \\ \mathsf{Vrfy}_{PK}(m, \sigma) = 1 \wedge m \notin \{m_1, \ldots, m_\ell\}\right],$$

where $m_1, \ldots, m_\ell$ are the messages that A submitted to $\mathsf{Sign}_{SK}(\cdot)$.

Under this definition of security, a digital signature emulates (the ideal qualities of) a handwritten signature. The definition shows that a digital signature on a message or document is easily verifiable by any recipient who knows the signer's public key; furthermore, a secure signature scheme is unforgeable in the sense that a third party cannot affix someone else's signature to a document without the signer's agreement.

Signature schemes also possess the important quality of *non-repudiation*; namely, a signer who has digitally signed a message cannot later deny doing so (of course, he can claim that his secret key was stolen or otherwise illegally obtained). Note that this property is not shared by MACs, because a tag on a given message could have been generated by either of the parties who share the secret key. Signatures, on the other hand, uniquely bind one party to the signed document.

It will be instructive to first look at a simple proposal of a signature scheme based on the RSA assumption, which is *not secure*. Unfortunately, this scheme is presented in many textbooks as a secure implementation of a signature scheme; hence, we refer to the scheme as the "textbook RSA scheme." Here, key generation involves choosing two large primes p, q of equal length and computing $N = pq$. Next, choose $e < N$ which is relatively prime to $\varphi(N)$ and compute d such that $ed = 1 \bmod \varphi(N)$. The public key is (N, e) and the secret key is (N, d). To sign a message $m \in \mathbb{Z}_N^*$, the signer computes

$$\sigma = m^d \bmod N;$$

verification of signature σ on message m is performed by checking that

$$\sigma^e \stackrel{?}{=} m \bmod N.$$

That this is indeed a signature scheme follows from the fact that $(m^d)^e = m^{de} = m \bmod N$ (see Section 9.3.2). What can we say about the security of the scheme?

*Historically, the definition of security for MACs was based on the earlier definition of security for signatures.

It is not hard to see that the textbook RSA scheme is completely insecure! An adversary can forge a valid message/signature pair as follows: choose arbitrary $\sigma \in \mathbb{Z}_N^*$ and set $m = \sigma^e \bmod N$. It is clear that the verification algorithm accepts σ as a valid signature on m.

In the previous attack, the adversary generates a signature on an essentially random message m. Here, we show how an adversary can forge a signature on a *particular* message m. First, the adversary finds arbitrary m_1, m_2 such that $m_1 m_2 = m \bmod N$; the adversary then requests and obtains signatures σ_1, σ_2 on m_1, m_2, respectively (recall that this is allowed by Definition 9.12). Now we claim that the verification algorithm accepts $\sigma = \sigma_1\sigma_2 \bmod N$ as a valid signature on m. Indeed:

$$(\sigma_1\sigma_2)^e = \sigma_1^e\sigma_2^e = m_1 m_2 = m \bmod N.$$

The two preceding examples illustrate that textbook RSA is *not* secure. The general approach, however, may be secure if the message is hashed (using a cryptographic hash function) before signing; this approach yields the *full-domain hash* (FDH) signature scheme [5]. In more detail, let $H : \{0,1\}^* \to \mathbb{Z}_N^*$ be a cryptographic hash function that might be included as part of the signer's public key. Now, message m is signed by computing $\sigma = H(m)^d \bmod N$; a signature σ on message m is verified by checking that $\sigma^e \stackrel{?}{=} H(m) \bmod N$. The presence of the hash (assuming a "good" hash function) prevents the two attacks mentioned above: for example, an adversary will still be able to generate σ, m' with $\sigma^e = m' \bmod N$ as before, but now the adversary will *not* be able to find a message m for which $H(m) = m'$. Similarly, the second attack is foiled because it is is difficult for an adversary to find m_1, m_2, m with $H(m_1)H(m_2) = H(m) \bmod N$. The use of the hash H has the additional advantage that messages of arbitrary length can now be signed.

It is, in fact, possible to prove the security of the FDH signature scheme based on the assumption that RSA is a trapdoor permutation and a (somewhat non-standard) assumption about the hash function H; however, it is beyond the scope of this work to discuss the necessary assumptions on H in order to enable a proof of security. We refer the interested reader to [5] for further details.

The Digital Signature Algorithm (DSA) (also known as the Digital Signature Standard [DSS]) [2, 20] is another widely used and standardized signature scheme whose security is related to the hardness of computing discrete logarithms (and which therefore offers an alternative to schemes whose security is based on, e.g., the RSA problem). Let p, q be primes such that $|q| = 160$ and q divides $p - 1$; typically, we might have $|p| = 512$. Let g be an element of order q in the multiplicative group $\mathbb{Z}_p^*$, and let $\langle g \rangle$ denote the subgroup of $\mathbb{Z}_p^*$ generated by g. Finally, let $H : \{0,1\}^* \to \{0,1\}^{160}$ be a cryptographic hash function. Parameters (p, q, g, H) are public, and can be shared by multiple signers. A signer's personal key is computed by choosing a random $x \in \mathbb{Z}_q$ and setting $y = g^x \bmod p$; the signer's public key is y and their private key is x. (Note that if computing discrete logarithms in $\langle g \rangle$ were easy, then it would be possible to compute a signer's secret key from their public key and the scheme would immediately be insecure.)

To sign a message $m \in \{0,1\}^*$ using secret key x, the signer generates a random $k \in \mathbb{Z}_q$ and computes

$$r = (g^k \bmod p) \bmod q$$
$$s = (H(m) + xr)k^{-1} \bmod q$$

The signature is (r, s). Verification of signature (r, s) on message m with respect to public key y is done by checking that $r, s \in \mathbb{Z}_q^*$ and

$$r \stackrel{?}{=} (g^{H(m)s^{-1}} y^{rs^{-1}} \bmod p) \bmod q.$$

It can be easily verified that signatures produced by the legitimate signer are accepted (with all but negligible probability) by the verification algorithm.

It is beyond the scope of this work to discuss the security of DSA; we refer the reader to a recent survey article [53] for further discussion and details.

Finally, we state the following result, which is of great theoretical importance but (unfortunately) of limited practical value.

Theorem 9.9 ([35, 40, 46]) *If there exists a one-way function family $\mathcal{F}$, then there exists a digital signature scheme secure against adaptive chosen-message attack.*

Defining Terms

Block cipher: An efficient instantiation of a pseudorandom function.
Ciphertext: The result of encrypting a message.
Collision-resistant hash function: Hash function for which it is infeasible to find two different inputs mapping to the same output.
Data integrity: Ensuring that modifications to a communicated message are detected.
Data secrecy: Hiding the contents of a communicated message.
Decrypt: To recover the original message from the transmitted ciphertext.
Digital signature scheme: Method for protecting data integrity in the public-key setting.
Encrypt: To apply an encryption scheme to a plaintext message.
Message-authentication code: Algorithm preserving data integrity in the private-key setting.
Mode of encryption: A method for using a block cipher to encrypt arbitrary-length messages.
One-time pad: A private-key encryption scheme achieving perfect secrecy.
One-way function: A function that is "easy" to compute but "hard" to invert.
Plaintext: The communicated data, or message.
Private-key encryption: Technique for ensuring data secrecy in the private-key setting.
Private-key setting: Setting in which communicating parties secretly share keys in advance of their communication.
Pseudorandom function: A keyed function that is indistinguishable from a truly random function.
Pseudorandom generator: A deterministic function that converts a short, random string to a longer, pseudorandom string.
Public-key encryption: Technique for ensuring data secrecy in the public-key setting.
Public-key setting: Setting in which parties generate public/private keys and widely disseminate their public keys.
Trapdoor permutation: A one-way permutation that is "easy" to invert if some trapdoor information is known.

References

[1] Alexi, W.B., Chor, B., Goldreich, O., and Schnorr, C.P. 1988. RSA/Rabin functions: certain parts are as hard as the whole. *SIAM J. Computing*, 17(2):194–209.

[2] ANSI X9.30. 1997. Public key cryptography for the financial services industry. Part 1: The digital signature algorithm (DSA). American National Standards Institute. American Bankers Association.

[3] Bellare, M., Desai, A., Jokipii, E., and Rogaway, P. 1997. A concrete security treatment of symmetric encryption. *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, IEEE, pp. 394–403.

[4] Bellare, M., Kilian, J., and Rogaway, P. 2000. The security of the cipher block chaining message authentication code. *J. of Computer and System Sciences*, 61(3):362–399.

[5] Bellare, M. and Rogaway, P. 1993. Random oracles are practical: a paradigm for designing efficient protocols. *First ACM Conference on Computer and Communications Security*, ACM, pp. 62–73.

[6] Bellare, M. and Rogaway, P. 1995. Optimal asymmetric encryption. *Advances in Cryptology — Eurocrypt '94*, Lecture Notes in Computer Science, Vol. 950, A. De Santis, Ed., Springer-Verlag, pp. 92–111.

[7] Bellare, M. and Rogaway, P. January 2003. Introduction to modern cryptography. Available at http://www.cs.ucsd.edu/users/mihir/cse207/classnotes.html.

[8] Black, J. and Rogaway, P. 2000. CBC MACs for arbitrary-length messages: the three-key constructions. *Advances in Cryptology — Crypto 2000*, Lecture Notes in Computer Science, Vol. 1880, M. Bellare, Ed., Springer-Verlag, pp. 197–215.

[9] Blum, M. and Micali, S. 1984. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM J. Computing*, 13(4):850–864.

[10] Boneh, D. 2001. Simplified OAEP for the RSA and Rabin functions. *Advances in Cryptology — Crypto 2001*, Lecture Notes in Computer Science, Vol. 2139, J. Kilian, Ed., Springer-Verlag, pp. 275–291.

[11] Childs, L.N. 2000. *A Concrete Introduction to Higher Algebra*. Springer-Verlag, Berlin.

[12] Cramer, R. and Shoup, V. 1998. A practical public-key cryptosystem provably secure against adaptive chosen ciphertext attack. *Advances in Cryptology — Crypto '98*, Lecture Notes in Computer Science, Vol. 1462, H. Krawczyk, Ed., Springer-Verlag, pp. 13–25.

[13] Daemen, J. and Rijmen, V. 2002. *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer-Verlag, Berlin.

[14] Delfs, H. and Knebl, H. 2002. *Introduction to Cryptography: Principles and Applications*. Springer-Verlag, Berlin.

[15] Diffie, W. and Hellman, M. 1976. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6): 644–654.

[16] El Gamal, T. 1985. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472.

[17] *Federal Information Processing Standards* publication #46. 1977. Data encryption standard. U.S. Department of Commerce/National Bureau of Standards.

[18] *Federal Information Processing Standards* publication #81. 1980. DES modes of operation. U.S. Department of Commerce/National Bureau of Standards.

[19] *Federal Information Processing Standards* Publication #180-1. 1995. Secure hash standard. U.S. Department of Commerce/National Institute of Standards and Technology.

[20] *Federal Information Processing Standards* Publication #186-2. 2000. Digital signature standard (DSS). U.S. Department of Commerce/National Institute of Standards and Technology.

[21] Fischlin, R. and Schnorr, C.P. 2000. Stronger security proofs for RSA and Rabin bits. *J. Cryptology*, 13(2):221–244.

[22] Fujisaki, E., Okamoto, T., Pointcheval, D., and Stern, J. 2001. RSA-OAEP is secure under the RSA assumption. *Advances in Cryptology — Crypto 2001*, Lecture Notes in Computer Science, Vol. 2139, J. Kilian, Ed., Springer-Verlag, pp. 260–274.

[23] Goldreich, O. 2001. *Foundations of Cryptography: Basic Tools*. Cambridge University Press.

[24] Goldreich, O. Foundations of cryptography, Vol. 2: basic applications. Available at http://www.wisdom.weizmann.ac.il/~oded/foc-vol2.html.

[25] Goldreich, O., Goldwasser, S., and Micali, S. 1986. How to construct random functions. *Journal of the ACM*, 33(4):792–807.

[26] Goldreich, O., Goldwasser, S., and Micali, S. 1985. On the cryptographic applications of random functions. *Advances in Cryptology — Crypto '84*, Lecture Notes in Computer Science, Vol. 196, G.R. Blakley and D. Chaum, Eds., Springer-Verlag, pp. 276–288.

[27] Goldreich, O. and Levin, L. 1989. Hard-core predicates for any one-way function. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, ACM, pp. 25–32.

[28] Goldwasser, S. and Bellare, M. August, 2001. Lecture notes on cryptography. Available at http://www.cs.ucsd.edu/users/mihir/papers/gb.html.

[29] Goldwasser, S. and Micali, S. 1984. Probabilistic encryption. *J. Computer and System Sciences*, 28(2):270–299.

[30] Goldwasser, S., Micali, S., and Rivest, R. 1988. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308.

[31] Håstad, J., Impagliazzo, R., Levin, L., and Luby, M. 1999. A pseudorandom generator from any one-way function. *SIAM J. Computing*, 28(4):1364–1396.

[32] Håstad, J., Schrift, A.W., and Shamir, A. 1993. The discrete logarithm modulo a composite hides $O(n)$ bits. *J. Computer and System Sciences*, 47(3):376–404.
[33] Knuth, D.E. 1997. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms (third edition)*. Addison-Wesley Publishing Company.
[34] Koblitz, N. 1999. *Algebraic Aspects of Cryptography*. Springer-Verlag, Berlin.
[35] Lamport, L. 1979. Constructing digital signatures from any one-way function. Technical Report CSL-98, SRI International, Palo Alto.
[36] Long, D.L. and Wigderson, A. 1988. The discrete logarithm problem hides $O(\log n)$ bits. *SIAM J. Computing*, 17(2):363–372.
[37] Luby, M. and Rackoff, C. 1988. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Computing*, 17(2):412–426.
[38] Menezes, A.J., van Oorschot, P.C., and Vanstone, S.A. 2001. *Handbook of Applied Cryptography*. CRC Press.
[39] Merkle, R. and Hellman, M. 1978. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24:525–530.
[40] Naor, M. and Yung, M. 1989. Universal one-way hash functions and their cryptographic applications. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, ACM, pp. 33–43.
[41] Petrank, E. and Rackoff, C. 2000. CBC MAC for real-time data sources. *J. of Cryptology*, 13(3): 315–338.
[42] Rabin, M.O. 1979. Digitalized signatures and public key functions as intractable as factoring. MIT/LCS/TR-212, MIT Laboratory for Computer Science.
[43] Rivest, R. 1990. Cryptography. Chapter 13 of *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, J. van Leeuwen, Ed., MIT Press.
[44] Rivest, R. 1992. The MD5 message-digest algorithm. RFC 1321, available at ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt.
[45] Rivest, R., Shamir, A., and Adleman, L.M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.
[46] Rompel, J. 1990. One-way functions are necessary and sufficient for secure signatures. *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM, pp. 387–394.
[47] Schneier, B. 1995. *Applied Cryptography: Protocols, Algorithms, and Source Code in C (second edition)*. John Wiley & Sons.
[48] Shoup, V. 2001. OAEP reconsidered. *Advances in Cryptology — Crypto 2001*, Lecture Notes in Computer Science, Vol. 2139, J. Kilian, Ed., Springer-Verlag, pp. 239–259.
[49] Simon, D. 1998. Finding collisions on a one-way street: can secure hash functions be based on general assumptions? *Advances in Cryptology — Eurocrypt '98*, Lecture Notes in Computer Science, Vol. 1403, K. Nyberg, Ed., Springer-Verlag, pp. 334–345.
[50] Sipser, M. 1996. *Introduction to the Theory of Computation*. Brooks/Cole Publishing Company.
[51] Stinson, D.R. 2002. *Cryptography: Theory and Practice (second edition)*. Chapman & Hall.
[52] Tsiounis, Y. and Yung, M. 1998. On the security of El Gamal based encryption. *Public Key Cryptography — PKC '98*, Lecture Notes in Computer Science, Vol. 1431, H. Imai and Y. Zheng, Eds., Springer-Verlag, pp. 117–134.
[53] Vaudenay, S. 2003. The security of DSA and ECDSA. *Public-Key Cryptography — PKC 2003*, Lecture Notes in Computer Science, Vol. 2567, Y. Desmedt, Ed., Springer-Verlag, pp. 309–323.
[54] Yao, A.C. 1982. Theory and application of trapdoor functions. *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, IEEE, pp. 80–91.

Further Information

A number of excellent sources are available for the reader interested in more information about modern cryptography. An excellent and enjoyable review of the field up to 1990 is given by Rivest [43]. Details on the more practical aspects of cryptography appear in the approachable textbooks of Stinson [51] and Schneier [47]; the latter also includes detail on implementing many popular cryptographic algorithms.

More formal and mathematical approaches to the subject (of which the present treatment is an example) are available in a number of well-written textbooks and online texts, including those by Goldwasser and Bellare [28], Goldreich [23, 24], Delfs and Knebl [14], and Bellare and Rogaway [7]. We also mention the comprehensive reference book by Menezes, van Oorschot, and Vanstone [38].

The International Association for Cryptologic Research (IACR) sponsors a number of conferences covering all areas of cryptography, with Crypto and Eurocrypt being perhaps the best known. Proceedings of these conferences (dating, in some cases, to the early 1980s) are published as part of Springer-Verlag's *Lecture Notes in Computer Science*. Research in theoretical cryptography often appears at the ACM Symposium on Theory of Computing, the Annual Symposium on Foundations of Computer Science (sponsored by IEEE), and elsewhere; more practice-oriented aspects of cryptography are covered in many security conferences, including the ACM Conference on Computer and Communications Security.

The IACR publishes the *Journal of Cryptology*, which is devoted exclusively to cryptography. Articles on cryptography frequently appear in the *Journal of Computer and System Sciences*, the *Journal of the ACM*, and the *SIAM Journal of Computing*.

10
Parallel Algorithms

Guy E. Blelloch
Carnegie Mellon University

Bruce M. Maggs
Carnegie Mellon University

10.1 Introduction

The subject of this chapter is the design and analysis of parallel algorithms. Most of today's computer algorithms are sequential, that is, they specify a sequence of steps in which each step consists of a single operation. As it has become more difficult to improve the performance of sequential computers, however, researchers have sought performance improvements in another place: parallelism. In contrast to a sequential algorithm, a parallel algorithm may perform multiple operations in a single step. For example, consider the problem of computing the sum of a sequence, A, of n numbers. The standard sequential algorithm computes the sum by making a single pass through the sequence, keeping a running sum of the numbers seen so far. It is not difficult, however, to devise an algorithm for computing the sum that performs many operations in parallel. For example, suppose that, in parallel, each element of A with an even index is paired and summed with the next element of A, which has an odd index, i.e., $A[0]$ is paired with $A[1]$, $A[2]$ with $A[3]$, and so on. The result is a new sequence of $\lceil n/2 \rceil$ numbers whose sum is identical to the sum that we wish to compute. This pairing and summing step can be repeated, and after $\lceil \log_2 n \rceil$ steps, only the final sum remains.

The parallelism in an algorithm can yield improved performance on many different kinds of computers. For example, on a parallel computer, the operations in a parallel algorithm can be performed simultaneously by different processors. Furthermore, even on a single-processor computer it is possible to exploit the parallelism in an algorithm by using multiple functional units, pipelined functional units, or pipelined memory systems. As these examples show, it is important to make a distinction between the parallelism in an algorithm and the ability of any particular computer to perform multiple operations in parallel. Typically, a parallel algorithm will run efficiently on a computer if the algorithm contains at least as much parallelism as the computer. Thus, good parallel algorithms generally can be expected to run efficiently on sequential computers as well as on parallel computers.

The remainder of this chapter consists of eight sections. Section 10.2 begins with a discussion of how to model parallel computers. Next, in Section 10.3 we cover some general techniques that have proven useful in the design of parallel algorithms. Section 10.4 to Section 10.8 present algorithms for solving problems from different domains. We conclude in Section 10.9 with a brief discussion of parallel complexity theory. Throughout this chapter, we assume that the reader has some familiarity with sequential algorithms and asymptotic analysis.

10.2 Modeling Parallel Computations

To analyze parallel algorithms it is necessary to have a formal model in which to account for costs. The designer of a sequential algorithm typically formulates the algorithm using an abstract model of computation called a *random-access machine* (RAM) [Aho et al. 1974, ch. 1]. In this model, the machine consists of a single processor connected to a memory system. Each basic central processing unit (CPU) operation, including arithmetic operations, logical operations, and memory accesses, requires one time step. The designer's goal is to develop an algorithm with modest time and memory requirements. The random-access machine model allows the algorithm designer to ignore many of the details of the computer on which the algorithm ultimately will be executed, but it captures enough detail that the designer can predict with reasonable accuracy how the algorithm will perform.

Modeling parallel computations is more complicated than modeling sequential computations because in practice parallel computers tend to vary more in their organizations than do sequential computers. As a consequence, a large proportion of the research on parallel algorithms has gone into the question of modeling, and many debates have raged over what the *right* model is, or about how practical various models are. Although there has been no consensus on the right model, this research has yielded a better understanding of the relationships among the models. Any discussion of parallel algorithms requires some understanding of the various models and the relationships among them.

Parallel models can be broken into two main classes: **multiprocessor models** and **work-depth models**. In this section we discuss each and then discuss how they are related.

10.2.1 Multiprocessor Models

A multiprocessor model is a generalization of the sequential RAM model in which there is more than one processor. Multiprocessor models can be classified into three basic types: local memory machines, modular memory machines, and **parallel random-access machines (PRAMs)**. Figure 10.1 illustrates the structures of these machines. A local memory machine consists of a set of n processors, each with its own local memory. These processors are attached to a common communication network. A modular memory machine consists of m memory modules and n processors all attached to a common network. A PRAM consists of a set of n processors all connected to a common shared memory [Fortune and Wyllie 1978, Goldshlager 1978, Savitch and Stimson 1979].

The three types of multiprocessors differ in the way memory can be accessed. In a local memory machine, each processor can access its own local memory directly, but it can access the memory in another processor only by sending a memory request through the network. As in the RAM model, all local operations, including local memory accesses, take unit time. The time taken to access the memory in another processor,

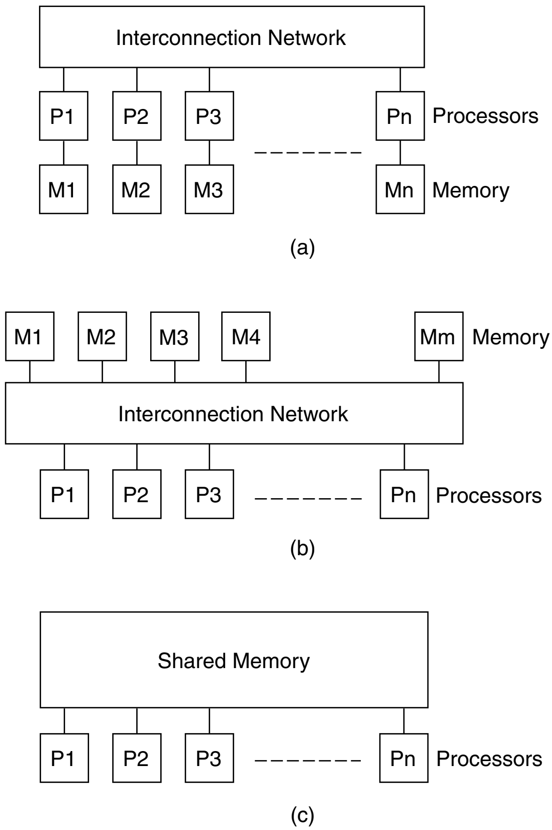


FIGURE 10.1 The three classes of multiprocessor machine models: (a) a local memory machine, (b) a modular memory machine, and (c) a parallel random-access machine (PRAM).

however, will depend on both the capabilities of the communication network and the pattern of memory accesses made by other processors, since these other accesses could congest the network. In a modular memory machine, a processor accesses the memory in a memory module by sending a memory request through the network. Typically, the processors and memory modules are arranged so that the time for any processor to access any memory module is roughly uniform. As in a local memory machine, the exact amount of time depends on the communication network and the memory access pattern. In a PRAM, in a single step each processor can simultaneously access any word of the memory by issuing a memory request directly to the shared memory.

The PRAM model is controversial because no real machine lives up to its ideal of unit-time access to shared memory. It is worth noting, however, that the ultimate purpose of an abstract model is not to directly model a real machine but to help the algorithm designer produce efficient algorithms. Thus, if an algorithm designed for a PRAM (or any other model) can be translated to an algorithm that runs efficiently on a real computer, then the model has succeeded. Later in this section, we show how algorithms designed for one parallel machine model can be translated so that they execute efficiently on another model.

The three types of multiprocessor models that we have defined are very broad, and these models further differ in network topology, network functionality, control, synchronization, and cache coherence. Many of these issues are discussed elsewhere in this volume. Here we will briefly discuss some of them.

10.2.1.1 Network Topology

A network is a collection of switches connected by communication channels. A processor or memory module has one or more communication ports that are connected to these switches by communication

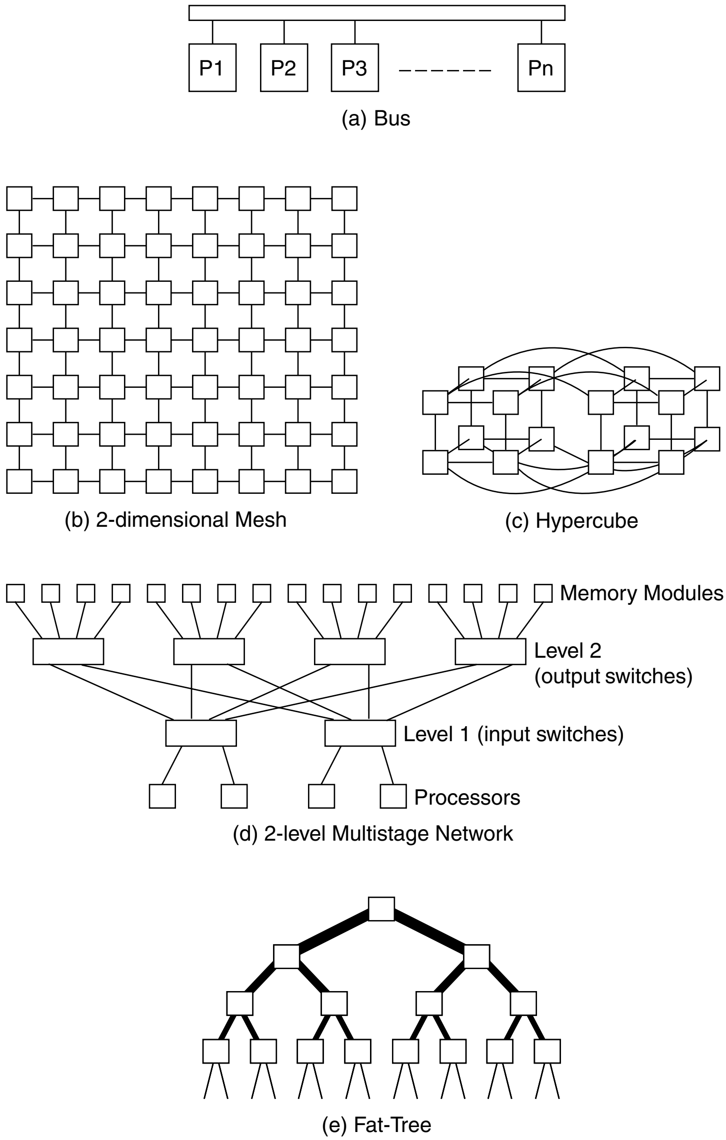


FIGURE 10.2 Various network topologies: (a) bus, (b) two-dimensional mesh, (c) hypercube, (d) two-level multistage network, and (e) fat-tree.

channels. The pattern of interconnection of the switches is called the network topology. The topology of a network has a large influence on the performance and also on the cost and difficulty of constructing the network. Figure 10.2 illustrates several different topologies.

The simplest network topology is a bus. This network can be used in both local memory machines and modular memory machines. In either case, all processors and memory modules are typically connected to a single bus. In each step, at most one piece of data can be written onto the bus. This datum might be a request from a processor to read or write a memory value, or it might be the response from the processor or memory module that holds the value. In practice, the advantages of using buses are that they are simple to build, and, because all processors and memory modules can observe the traffic on a bus, it is relatively easy to develop protocols that allow processors to cache memory values locally.

The disadvantage of using a bus is that the processors have to take turns accessing the bus. Hence, as more processors are added to a bus, the average time to perform a memory access grows proportionately.

A two-dimensional *mesh* is a network that can be laid out in a rectangular fashion. Each switch in a mesh has a distinct label (x, y) where $0 \le x \le X - 1$ and $0 \le y \le Y - 1$. The values X and Y determine the length of the sides of the mesh. The number of switches in a mesh is thus $X \cdot Y$. Every switch, except those on the sides of the mesh, is connected to four neighbors: one to the north, one to the south, one to the east, and one to the west. Thus, a switch labeled (x, y), where $0 < x < X - 1$ and $0 < y < Y - 1$ is connected to switches $(x, y + 1)$, $(x, y - 1)$, $(x + 1, y)$, and $(x - 1, y)$. This network typically appears in a local memory machine, i.e., a processor along with its local memory is connected to each switch, and remote memory accesses are made by routing messages through the mesh. Figure 10.2b shows an example of an 8×8 mesh.

Several variations on meshes are also popular, including three-dimensional meshes, toruses, and hypercubes. A *torus* is a mesh in which the switches on the sides have connections to the switches on the opposite sides. Thus, every switch (x, y) is connected to four other switches: $(x, y + 1 \bmod Y)$, $(x, y - 1 \bmod Y)$, $(x + 1 \bmod X, y)$, and $(x - 1 \bmod X, y)$. A hypercube is a network with 2^n switches in which each switch has a distinct n-bit label. Two switches are connected by a communication channel in a hypercube if their labels differ in precisely one-bit position.

A *multistage network* is used to connect one set of switches called the *input switches* to another set called the *output switches* through a sequence of stages of switches. Such networks were originally designed for telephone networks [Beneš 1965]. The stages of a multistage network are numbered 1 through L, where L is the **depth** of the network. The input switches form stage 1 and the output switches form stage L. In most multistage networks, it is possible to send a message from any input switch to any output switch along a path that traverses the stages of the network in order from 1 to L. Multistage networks are frequently used in modular memory computers; typically, processors are attached to input switches, and memory modules to output switches. There are many different multistage network topologies. Figure 10.2d, for example, shows a 2-stage network that connects 4 processors to 16 memory modules. Each switch in this network has two channels at the bottom and four channels at the top. The ratio of processors to memory modules in this example is chosen to reflect the fact that, in practice, a processor is capable of generating memory access requests faster than a memory module is capable of servicing them.

A *fat-tree* is a network whose overall structure is that of a tree [Leiserson 1985]. Each edge of the tree, however, may represent many communication channels, and each node may represent many network switches (hence the name fat). Figure 10.2e shows a fat-tree whose overall structure is that of a binary tree. Typically the capacities of the edges near the root of the tree are much larger than the capacities near the leaves. For example, in this tree the two edges incident on the root represent 8 channels each, whereas the edges incident on the leaves represent only 1 channel each. One way to construct a local memory machine is to connect a processor along with its local memory to each leaf of the fat-tree. In this scheme, a message from one processor to another first travels up the tree to the least common ancestor of the two processors and then down the tree.

Many algorithms have been designed to run efficiently on particular network topologies such as the mesh or the hypercube. For an extensive treatment such algorithms, see Leighton [1992]. Although this approach can lead to very fine-tuned algorithms, it has some disadvantages. First, algorithms designed for one network may not perform well on other networks. Hence, in order to solve a problem on a new machine, it may be necessary to design a new algorithm from scratch. Second, algorithms that take advantage of a particular network tend to be more complicated than algorithms designed for more abstract models such as the PRAM because they must incorporate some of the details of the network. Nevertheless, there are some operations that are performed so frequently by a parallel machine that it makes sense to design a fine-tuned network-specific algorithm. For example, the algorithm that routes messages or memory access requests through the network should exploit the network topology. Other examples include algorithms for broadcasting a message from one processor to many other processors, for

collecting the results computed in many processors in a single processor, and for synchronizing processors.

An alternative to modeling the topology of a network is to summarize its routing capabilities in terms of two parameters, its latency and bandwidth. The latency L of a network is the time it takes for a message to traverse the network. In actual networks this will depend on the topology of the network, which particular ports the message is passing between, and the congestion of messages in the network. The latency, however, often can be usefully modeled by considering the worst-case time assuming that the network is not heavily congested. The bandwidth at each port of the network is the rate at which a processor can inject data into the network. In actual networks this will depend on the topology of the network, the bandwidths of the network's individual communication channels, and, again, the congestion of messages in the network. The bandwidth often can be usefully modeled as the maximum rate at which processors can inject messages into the network without causing it to become heavily congested, assuming a uniform distribution of message destinations. In this case, the bandwidth can be expressed as the minimum *gap* g between successive injections of messages into the network.

Three models that characterize a network in terms of its latency and bandwidth are the postal model [Bar-Noy and Kipnis 1992], the bulk-synchronous parallel (BSP) model [Valiant 1990a], and the LogP model [Culler et al. 1993]. In the postal model, a network is described by a single parameter, L, its latency. The bulk-synchronous parallel model adds a second parameter, g, the minimum ratio of computation steps to communication steps, i.e., the gap. The LogP model includes both of these parameters and adds a third parameter, o, the overhead, or wasted time, incurred by a processor upon sending or receiving a message.

10.2.1.2 Primitive Operations

As well as specifying the general form of a machine and the network topology, we need to define what operations the machine supports. We assume that all processors can perform the same instructions as a typical processor in a sequential machine. In addition, processors may have special instructions for issuing nonlocal memory requests, for sending messages to other processors, and for executing various global operations, such as synchronization. There can also be restrictions on when processors can simultaneously issue instructions involving nonlocal operations. For example a machine might not allow two processors to write to the same memory location at the same time. The particular set of instructions that the processors can execute may have a large impact on the performance of a machine on any given algorithm. It is therefore important to understand what instructions are supported before one can design or analyze a parallel algorithm. In this section we consider three classes of nonlocal instructions: (1) how global memory requests interact, (2) synchronization, and (3) global operations on data.

When multiple processors simultaneously make a request to read or write to the same resource — such as a processor, memory module, or memory location — there are several possible outcomes. Some machine models simply forbid such operations, declaring that it is an error if more than one processor tries to access a resource simultaneously. In this case we say that the machine allows only *exclusive* access to the resource. For example, a PRAM might allow only exclusive read or write access to each memory location. A PRAM of this type is called an **exclusive-read exclusive-write (EREW)** PRAM. Other machine models may allow unlimited access to a shared resource. In this case we say that the machine allows *concurrent* access to the resource. For example, a **concurrent-read concurrent-write (CRCW)** PRAM allows both concurrent read and write access to memory locations, and a **CREW** PRAM allows **concurrent reads but only exclusive writes**. When making a concurrent write to a resource such as a memory location there are many ways to resolve the conflict. Some possibilities are to choose an arbitrary value from those written, to choose the value from the processor with the lowest index, or to take the *logical or* of the values written. A final choice is to allow for *queued* access, in which case concurrent access is permitted but the time for a step is proportional to the maximum number of accesses to any resource. A queue-read queue-write (QRQW) PRAM allows for such accesses [Gibbons et al. 1994].

In addition to reads and writes to nonlocal memory or other processors, there are other important primitives that a machine may supply. One class of such primitives supports synchronization. There are a variety of different types of synchronization operations and their costs vary from model to model. In the PRAM model, for example, it is assumed that all processors operate in lock step, which provides implicit synchronization. In a local-memory machine the cost of synchronization may be a function of the particular network topology. Some machine models supply more powerful primitives that combine arithmetic operations with communication. Such operations include the prefix and **multiprefix** operations, which are defined in the subsections on scans and multiprefix and fetch-and-add.

10.2.2 Work-Depth Models

Because there are so many different ways to organize parallel computers, and hence to model them, it is difficult to select one multiprocessor model that is appropriate for all machines. The alternative to focusing on the machine is to focus on the algorithm. In this section we present a class of models called work-depth models. In a work-depth model, the cost of an algorithm is determined by examining the total number of operations that it performs and the dependencies among those operations. An algorithm's **work** W is the total number of operations that it performs; its *depth* D is the longest chain of dependencies among its operations. We call the ratio $\mathcal{P} = W/D$ the *parallelism* of the algorithm. We say that a parallel algorithm is work-efficient relative to a sequential algorithm if it does at most a constant factor more work.

The work-depth models are more abstract than the multiprocessor models. As we shall see, however, algorithms that are efficient in the work-depth models often can be translated to algorithms that are efficient in the multiprocessor models and from there to real parallel computers. The advantage of a work-depth model is that there are no machine-dependent details to complicate the design and analysis of algorithms. Here we consider three classes of work-depth models: circuit models, vector machine models, and language-based models. We will be using a language-based model in this chapter, and so we will return to these models later in this section.

The most abstract work-depth model is the *circuit model*. In this model, an algorithm is modeled as a family of directed acyclic circuits. There is a circuit for each possible size of the input. A circuit consists of nodes and arcs. A node represents a basic operation, such as adding two values. For each input to an operation (i.e., node), there is an incoming arc from another node or from an input to the circuit. Similarly, there are one or more outgoing arcs from each node representing the result of the operation. The work of a circuit is the total number of nodes. (The work is also called the *size*.) The depth of a circuit is the length of the longest directed path between any pair of nodes. Figure 10.3 shows a circuit in which the inputs are at the top, each + is an adder circuit, and each of the arcs carries the result of an adder circuit. The final

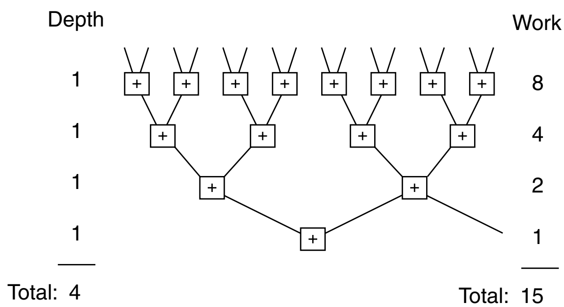


FIGURE 10.3 Summing 16 numbers on a tree. The total depth (longest chain of dependencies) is 4 and the total work (number of operations) is 15.

sum is returned at the bottom. Circuit models have been used for many years to study various theoretical aspects of parallelism, for example, to prove that certain problems are hard to solve in parallel (see Karp and Ramachandran [1990] for an overview).

In a *vector model*, an algorithm is expressed as a sequence of steps, each of which performs an operation on a vector (i.e., sequence) of input values, and produces a vector result [Pratt and Stockmeyer 1976, Blelloch 1990]. The work of each step is equal to the length of its input (or output) vector. The work of an algorithm is the sum of the work of its steps. The depth of an algorithm is the number of vector steps.

In a *language* model, a work-depth cost is associated with each programming language construct [Blelloch and Greiner 1995, Blelloch 1996]. For example, the work for calling two functions in parallel is equal to the sum of the work of the two calls. The depth, in this case, is equal to the maximum of the depth of the two calls.

10.2.3 Assigning Costs to Algorithms

In the work-depth models, the cost of an algorithm is determined by its work and by its depth. The notions of work and depth also can be defined for the multiprocessor models. The work W performed by an algorithm is equal to the number of processors times the time required for the algorithm to complete execution. The depth D is equal to the total time required to execute the algorithm.

The depth of an algorithm is important because there are some applications for which the time to perform a computation is crucial. For example, the results of a weather-forecasting program are useful only if the program completes execution before the weather does!

Generally, however, the most important measure of the cost of an algorithm is the work. This can be justified as follows. The cost of a computer is roughly proportional to the number of processors in the computer. The cost for purchasing time on a computer is proportional to the cost of the computer times the amount of time used. The total cost of performing a computation, therefore, is roughly proportional to the number of processors in the computer times the amount of time used, i.e., the work.

In many instances, the cost of running a computation on a parallel computer may be slightly larger than the cost of running the same computation on a sequential computer. If the time to completion is sufficiently improved, however, this extra cost often can be justified. As we shall see, in general there is a tradeoff between work and time to completion. It is rarely the case, however, that a user is willing to give up any more than a small constant factor in cost for an improvement in time.

10.2.4 Emulations Among Models

Although it may appear that a different algorithm must be designed for each of the many parallel models, there are often automatic and efficient techniques for translating algorithms designed for one model into algorithms designed for another. These translations are *work preserving* in the sense that the work performed by both algorithms is the same, to within a constant factor. For example, the following theorem, known as Brent's theorem [1974], shows that an algorithm designed for the circuit model can be translated in a work-preserving fashion to a PRAM algorithm.

Theorem 10.1 (Brent's theorem) *Any algorithm that can be expressed as a circuit of size (i.e., work) W and depth D in the circuit model can be executed in $O(W/P + D)$ steps in the PRAM model.*

Proof **10.1** The basic idea is to have the PRAM emulate the computation specified by the circuit in a level-by-level fashion. The level of a node is defined as follows. A node is on level 1 if all of its inputs are also inputs to the circuit. Inductively, the level of any other node is one greater than the maximum of the level of the nodes with arcs into it. Let l_i denote the number of nodes on level i. Then, by assigning $\lceil l_i/P \rceil$ operations to each of the P processors in the PRAM, the operations for level i can be performed

in $O(\lceil l_i/P \rceil)$ steps. Summing the time over all D levels, we have

$$\begin{aligned} T_{\text{PRAM}}(W, D, P) &= O\left(\sum_{i=1}^{D} \left\lceil \frac{l_i}{P} \right\rceil\right) \\ &= O\left(\sum_{i=1}^{D} \left(\frac{l_i}{P} + 1\right)\right) \\ &= O\left(\frac{1}{P}\left(\sum_{i=1}^{D} l_i\right) + D\right) \\ &= O\left(W/P + D\right) \end{aligned}$$

□

The total work performed by the PRAM, i.e., the processor-time product, is $O(W + PD)$. This emulation is work preserving to within a constant factor when the parallelism ($\mathcal{P} = W/D$) is at least as large as the number of processors P, in this case the work is $O(W)$. The requirement that the parallelism exceed the number of processors is typical of work-preserving emulations.

Brent's theorem shows that an algorithm designed for one of the work-depth models can be translated in a work-preserving fashion on to a multiprocessor model. Another important class of work-preserving translations is those that translate between different multiprocessor models. The translation we consider here is the work-preserving translation of algorithms written for the PRAM model to algorithms for a more realistic machine model. In particular, we consider a *butterfly machine* in which P processors are attached through a butterfly network of depth $\log P$ to P memory banks. We assume that, in constant time, a processor can hash a virtual memory address to a physical memory bank and an address within that bank using a sufficiently powerful hash function. This scheme was first proposed by Karlin and Upfal [1988] for the EREW PRAM model. Ranade [1991] later presented a more general approach that allowed the butterfly to efficiently emulate CRCW algorithms.

Theorem 10.2 *Any algorithm that takes time T on a P-processor PRAM can be translated into an algorithm that takes time $O(T(P/P' + \log P'))$, with high probability, on a P'-processor butterfly machine.*

Sketch of proof Each of the P' processors in the butterfly machine emulates a set of P/P' PRAM processors. The butterfly machine emulates the PRAM in a step-by-step fashion. First, each butterfly processor emulates one step of each of its P/P' PRAM processors. Some of the PRAM processors may wish to perform memory accesses. For each memory access, the butterfly processor hashes the memory address to a physical memory bank and an address within the bank and then routes a message through the network to that bank. These messages are pipelined so that a processor can have multiple outstanding requests. Ranade proved that if each processor in a P-processor butterfly machine sends at most P/P' messages whose destinations are determined by a sufficiently powerful hash function, then the network can deliver all of the messages, along with responses, in $O(P/P' + \log P')$ time. The $\log P'$ term accounts for the latency of the network and for the fact that there will be some congestion at memory banks, even if each processor sends only a single message.

This theorem implies that, as long as $P \geq P' \log P'$, i.e., if the number of processors employed by the PRAM algorithm exceeds the number of processors in the butterfly machine by a factor of at least $\log P'$, then the emulation is work preserving. When translating algorithms from a guest multiprocessor model (e.g., the PRAM) to a host multiprocessor model (e.g., the butterfly machine), it is not uncommon to require that the number of guest processors exceed the number of host processors by a factor proportional to the latency of the host. Indeed, the latency of the host often can be hidden by giving it a larger guest to emulate. If the bandwidth of the host is smaller than the bandwidth of a comparably sized guest, however, it usually is much more difficult for the host to perform a work-preserving emulation of the guest.

For more information on PRAM emulations, the reader is referred to Harris [1994] and Valiant [1990].

10.2.5 Model Used in This Chapter

Because there are so many work-preserving translations between different parallel models of computation, we have the luxury of choosing the model that we feel most clearly illustrates the basic ideas behind the algorithms, a work-depth language model. Here we define the model we will use in this chapter in terms of a set of language constructs and a set of rules for assigning costs to the constructs. The description we give here is somewhat informal, but it should suffice for the purpose of this chapter. The language and costs can be properly formalized using a profiling semantics [Blelloch and Greiner 1995].

Most of the syntax that we use should be familiar to readers who have programmed in Algol-like languages, such as Pascal and C. The constructs for expressing parallelism, however, may be unfamiliar. We will be using two parallel constructs — a parallel *apply-to-each* construct and a *parallel-do* construct — and a small set of parallel primitives on sequences (one-dimensional arrays). Our language constructs, syntax, and cost rules are based on the NESL language [Blelloch 1996].

The apply-to-each construct is used to apply an expression over a sequence of values in parallel. It uses a setlike notation. For example, the expression

$$\{a * a : a \in [3, -4, -9, 5]\}$$

squares each element of the sequence $[3, -4, -9, 5]$ returning the sequence $[9, 16, 81, 25]$. This can be read: "in parallel, for each a in the sequence $[3, -4, -9, 5]$, square a." The apply-to-each construct also provides the ability to subselect elements of a sequence based on a filter. For example,

$$\{a * a : a \in [3, -4, -9, 5] \mid a > 0\}$$

can be read: "in parallel, for each a in the sequence $[3, -4, -9, 5]$ such that a is greater than 0, square a." It returns the sequence $[9, 25]$. The elements that remain maintain their relative order.

The parallel-do construct is used to evaluate multiple statements in parallel. It is expressed by listing the set of statements after an **in parallel do**. For example, the following fragment of code calls $\text{FUN1}(X)$ and assigns the result to A and in parallel calls $\text{FUN2}(Y)$ and assigns the result to B:

in parallel do

$$A := \text{FUN1}(X)$$
$$B := \text{FUN2}(Y)$$

The parallel-do completes when all the parallel subcalls complete.

Work and depth are assigned to our language constructs as follows. The work and depth of a scalar primitive operation is one. For example, the work and depth for evaluating an expression such as $3 + 4$ is one. The work for applying a function to every element in a sequence is equal to the sum of the work for each of the individual applications of the function. For example, the work for evaluating the expression

$$\{a * a : a \in [0..n)\}$$

which creates an n-element sequence consisting of the squares of 0 through $n - 1$, is n. The depth for applying a function to every element in a sequence is equal to the maximum of the depths of the individual applications of the function. Hence, the depth of the previous example is one. The work for a parallel-do construct is equal to the sum of the work for each of its statements. The depth is equal to the maximum depth of its statements. In all other cases, the work and depth for a sequence of operations is the sum of the work and depth for the individual operations.

In addition to the parallelism supplied by apply-to-each, we will use four built-in functions on sequences, *dist*, ++ (append), *flatten*, and ← (write), each of which can be implemented in parallel. The function *dist* creates a sequence of identical elements. For example, the expression *dist* (3, 5) creates the sequence

$$[3, 3, 3, 3, 3]$$

The $++$ function appends two sequences. For example, $[2,1]++[5,0,3]$ create the sequence $[2,1,5,0,3]$. The *flatten* function converts a nested sequence (a sequence for which each element is itself a sequence) into a flat sequence. For example,

$$\textit{flatten}([[3,5],[3,2],[1,5],[4,6]])$$

creates the sequence

$$[3,5,3,2,1,5,4,6]$$

The $\leftarrow$ function is used to write multiple elements into a sequence in parallel. It takes two arguments. The first argument is the sequence to modify and the second is a sequence of integer-value pairs that specify what to modify. For each pair (i, v), the value v is inserted into position i of the destination sequence. For example,

$$[0,0,0,0,0,0,0,0] \leftarrow [(4,-2),(2,5),(5,9)]$$

inserts the -2, 5, and 9 into the sequence at locations 4, 2, and 5, respectively, returning

$$[0,0,5,0,-2,9,0,0]$$

As in the PRAM model, the issue of concurrent writes arises if an index is repeated. Rather than choosing a single policy for resolving concurrent writes, we will explain the policy used for the individual algorithms. All of these functions have depth one and work n, where n is the size of the sequence(s) involved. In the case of the $\leftarrow$, the work is proportional to the length of the sequence of integer-value pairs, not the modified sequence, which might be much longer. In the case of $++$, the work is proportional to the length of the second sequence.

We will use a few shorthand notations for specifying sequences. The expression $[-2..1]$ specifies the same sequence as the expression $[-2,-1,0,1]$. Changing the left or right brackets surrounding a sequence omits the first or last elements, i.e., $[-2..1)$ denotes the sequence $[-2,-1,0]$. The notation $A[i..j]$ denotes the subsequence consisting of elements $A[i]$ through $A[j]$. Similarly, $A[i, j)$ denotes the subsequence $A[i]$ through $A[j-1]$. We will assume that sequence indices are zero based, i.e., $A[0]$ extracts the first element of the sequence A.

Throughout this chapter, our algorithms make use of random numbers. These numbers are generated using the functions *rand_bit()*, which returns a random bit, and *rand_int(h)*, which returns a random integer in the range $[0, h-1]$.

10.3 Parallel Algorithmic Techniques

As with sequential algorithms, in parallel algorithm design there are many general techniques that can be used across a variety of problem areas. Some of these are variants of standard sequential techniques, whereas others are new to parallel algorithms. In this section we introduce some of these techniques, including parallel divide-and-conquer, randomization, and parallel pointer manipulation. In later sections on algorithms we will make use of them.

10.3.1 Divide-and-Conquer

A divide-and-conquer algorithm first splits the problem to be solved into subproblems that are easier to solve than the original problem either because they are smaller instances of the original problem, or because they are different but easier problems. Next, the algorithm solves the subproblems, possibly recursively. Typically, the subproblems can be solved independently. Finally, the algorithm merges the solutions to the subproblems to construct a solution to the original problem.

The divide-and-conquer paradigm improves program modularity and often leads to simple and efficient algorithms. It has, therefore, proven to be a powerful tool for sequential algorithm designers. Divide-and-conquer plays an even more prominent role in parallel algorithm design. Because the subproblems created in the first step are typically independent, they can be solved in parallel. Often the subproblems are solved recursively and thus the next divide step yields even more subproblems to be solved in parallel. As a consequence, even divide-and-conquer algorithms that were designed for sequential machines typically have some inherent parallelism. Note, however, that in order for divide-and-conquer to yield a highly parallel algorithm, it often is necessary to parallelize the divide step and the merge step. It is also common in parallel algorithms to divide the original problem into as many subproblems as possible, so that they all can be solved in parallel.

As an example of parallel divide-and-conquer, consider the sequential mergesort algorithm. Mergesort takes a set of n keys as input and returns the keys in sorted order. It works by splitting the keys into two sets of $n/2$ keys, recursively sorting each set, and then merging the two sorted sequences of $n/2$ keys into a sorted sequence of n keys. To analyze the sequential running time of mergesort we note that two sorted sequences of $n/2$ keys can be merged in $O(n)$ time. Hence, the running time can be specified by the recurrence

$$T(n) = \begin{cases} 2T(n/2) + O(n) & n > 1 \\ O(1) & n = 1 \end{cases}$$

which has the solution $T(n) = O(n \log n)$. Although not designed as a parallel algorithm, mergesort has some inherent parallelism since the two recursive calls can be made in parallel. This can be expressed as:

Algorithm: MERGESORT(A).

```
if (|A| = 1) then return A
else
    in parallel do
        L := MERGESORT(A[0..|A|/2])
        R := MERGESORT(A[|A|/2..|A|])
    return MERGE(L, R)
```

Recall that in our work-depth model we can analyze the depth of an algorithm that makes parallel calls by taking the maximum depth of the two calls, and the work by taking the sum. We assume that the merging remains sequential so that the work and depth to merge two sorted sequences of $n/2$ keys is $O(n)$. Thus, for mergesort the work and depth are given by the recurrences:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ D(n) &= \max(D(n/2), D(n/2)) + O(n) \\ &= D(n/2) + O(n) \end{aligned}$$

As expected, the solution for the work is $W(n) = O(n \log n)$, i.e., the same as the time for the sequential algorithm. For the depth, however, the solution is $D(n) = O(n)$, which is smaller than the work. Recall that we defined the parallelism of an algorithm as the ratio of the work to the depth. Hence, the parallelism of this algorithm is $O(\log n)$ (not very much). The problem here is that the merge step remains sequential, and this is the bottleneck.

As mentioned earlier, the parallelism in a divide-and-conquer algorithm often can be enhanced by parallelizing the divide step and/or the merge step. Using a parallel merge [Shiloach and Vishkin 1982], two sorted sequences of $n/2$ keys can be merged with work $O(n)$ and depth $O(\log n)$. Using this merge

algorithm, the recurrence for the depth of mergesort becomes

$$D(n) = D(n/2) + O(\log n)$$

which has solution $D(n) = O(\log^2 n)$. Using a technique called **pipelined divide-and-conquer**, the depth of mergesort can be further reduced to $O(\log n)$ [Cole 1988]. The idea is to start the merge at the top level before the recursive calls complete.

Divide-and-conquer has proven to be one of the most powerful techniques for solving problems in parallel. In this chapter we will use it to solve problems from computational geometry, sorting, and performing fast Fourier transforms. Other applications range from linear systems to factoring large numbers to n-body simulations.

10.3.2 Randomization

The use of random numbers is ubiquitous in parallel algorithms. Intuitively, randomness is helpful because it allows processors to make local decisions which, with high probability, add up to good global decisions. Here we consider three uses of randomness.

10.3.2.1 Sampling

One use of randomness is to select a representative sample from a set of elements. Often, a problem can be solved by selecting a sample, solving the problem on that sample, and then using the solution for the sample to guide the solution for the original set. For example, suppose we want to sort a collection of integer keys. This can be accomplished by partitioning the keys into buckets and then sorting within each bucket. For this to work well, the buckets must represent nonoverlapping intervals of integer values and contain approximately the same number of keys. **Random sampling** is used to determine the boundaries of the intervals. First, each processor selects a random sample of its keys. Next, all of the selected keys are sorted together. Finally, these keys are used as the boundaries. Such random sampling also is used in many parallel computational geometry, graph, and string matching algorithms.

10.3.2.2 Symmetry Breaking

Another use of randomness is in **symmetry breaking**. For example, consider the problem of selecting a large independent set of vertices in a graph in parallel. (A set of vertices is *independent* if no two are neighbors.) Imagine that each vertex must decide, in parallel with all other vertices, whether to join the set or not. Hence, if one vertex chooses to join the set, then all of its neighbors must choose not to join the set. The choice is difficult to make simultaneously by each vertex if the local structure at each vertex is the same, for example, if each vertex has the same number of neighbors. As it turns out, the impasse can be resolved by using randomness to break the symmetry between the vertices [Luby 1985].

10.3.2.3 Load Balancing

A third use is load balancing. One way to quickly partition a large number of data items into a collection of approximately evenly sized subsets is to randomly assign each element to a subset. This technique works best when the average size of a subset is at least logarithmic in the size of the original set.

10.3.3 Parallel Pointer Techniques

Many of the traditional sequential techniques for manipulating lists, trees, and graphs do not translate easily into parallel techniques. For example, techniques such as traversing the elements of a linked list, visiting the nodes of a tree in postorder, or performing a depth-first traversal of a graph appear to be inherently sequential. Fortunately, these techniques often can be replaced by parallel techniques with roughly the same power.

10.3.3.1 Pointer Jumping

One of the earliest parallel pointer techniques is **pointer jumping** [Wyllie 1979]. This technique can be applied to either lists or trees. In each pointer jumping step, each node in parallel replaces its pointer with that of its successor (or parent). For example, one way to label each node of an n-node list (or tree) with the label of the last node (or root) is to use pointer jumping. After at most $\lceil \log n \rceil$ steps, every node points to the same node, the end of the list (or root of the tree). This is described in more detail in the subsection on pointer jumping.

10.3.3.2 Euler Tour

An Euler tour of a directed graph is a path through the graph in which every edge is traversed exactly once. In an undirected graph each edge is typically replaced with two oppositely directed edges. The Euler tour of an undirected tree follows the perimeter of the tree visiting each edge twice, once on the way down and once on the way up. By keeping a linked structure that represents the Euler tour of a tree, it is possible to compute many functions on the tree, such as the size of each subtree [Tarjan and Vishkin 1985]. This technique uses linear work and parallel depth that is independent of the depth of the tree. The Euler tour often can be used to replace standard traversals of a tree, such as a depth-first traversal.

10.3.3.3 Graph Contraction

Graph contraction is an operation in which a graph is reduced in size while maintaining some of its original structure. Typically, after performing a graph contraction operation, the problem is solved recursively on the contracted graph. The solution to the problem on the contracted graph is then used to form the final solution. For example, one way to partition a graph into its connected components is to first contract the graph by merging some of the vertices into their neighbors, then find the connected components of the contracted graph, and finally undo the contraction operation. Many problems can be solved by contracting trees [Miller and Reif 1989, 1991], in which case the technique is called **tree contraction**. More examples of graph contraction can be found in Section 10.5.

10.3.3.4 Ear Decomposition

An ear decomposition of a graph is a partition of its edges into an ordered collection of paths. The first path is a cycle, and the others are called ears. The endpoints of each ear are anchored on previous paths. Once an ear decomposition of a graph is found, it is not difficult to determine if two edges lie on a common cycle. This information can be used in algorithms for determining biconnectivity, triconnectivity, 4-connectivity, and planarity [Maon et al. 1986, Miller and Ramachandran 1992]. An ear decomposition can be found in parallel using linear work and logarithmic depth, independent of the structure of the graph. Hence, this technique can be used to replace the standard sequential technique for solving these problems, depth-first search.

10.3.4 Other Techniques

Many other techniques have proven to be useful in the design of parallel algorithms. Finding small graph separators is useful for partitioning data among processors to reduce communication [Reif 1993, ch. 14]. Hashing is useful for load balancing and mapping addresses to memory [Vishkin 1984, Karlin and Upfal 1988]. Iterative techniques are useful as a replacement for direct methods for solving linear systems [Bertsekas and Tsitsiklis 1989].

10.4 Basic Operations on Sequences, Lists, and Trees

We begin our presentation of parallel algorithms with a collection of algorithms for performing basic operations on sequences, lists, and trees. These operations will be used as subroutines in the algorithms that follow in later sections.

10.4.1 Sums

As explained at the opening of this chapter, there is a simple recursive algorithm for computing the sum of the elements in an array:

Algorithm: SUM(A).

if $|A| = 1$ **then return** $A[0]$
else return SUM($\{A[2i] + A[2i+1] : i \in [0..|A|/2)\}$)

The work and depth for this algorithm are given by the recurrences

$$W(n) = W(n/2) + O(n) = O(n)$$
$$D(n) = D(n/2) + O(1) = O(\log n)$$

which have solutions $W(n) = O(n)$ and $D(n) = O(\log n)$. This algorithm also can be expressed without recursion (using a **while** loop), but the recursive version forshadows the recursive algorithm for implementing the **scan** function.

As written, the algorithm works only on sequences that have lengths equal to powers of 2. Removing this restriction is not difficult by checking if the sequence is of odd length and separately adding the last element in if it is. This algorithm also can easily be modified to compute the sum relative to any associative operator in place of $+$. For example, the use of max would return the maximum value of a sequence.

10.4.2 Scans

The *plus-scan* operation (also called **all-prefix-sums**) takes a sequence of values and returns a sequence of equal length for which each element is the sum of all previous elements in the original sequence. For example, executing a plus-scan on the sequence $[3, 5, 3, 1, 6]$ returns $[0, 3, 8, 11, 12]$. The scan operation can be implemented by the following algorithm [Stone 1975]:

Algorithm: SCAN(A).

1 **if** $|A| = 1$ **then return** $[0]$
2 **else**
3 $\quad S =$ SCAN($\{A[2i] + A[2i+1] : i \in [0..|A|/2)\}$)
4 $\quad R = \{$**if** $(i \bmod 2) = 0$ **then** $S[i/2]$ **else** $S[(i-1)/2] + A[i-1] : i \in [0..|A|)\}$
5 **return** R

The algorithm works by elementwise adding the even indexed elements of A to the odd indexed elements of A and then recursively solving the problem on the resulting sequence (line 3). The result S of the recursive call gives the plus-scan values for the even positions in the output sequence R. The value for each of the odd positions in R is simply the value for the preceding even position in R plus the value of the preceding position from A.

The asymptotic work and depth costs of this algorithm are the same as for the SUM operation, $W(n) = O(n)$ and $D(n) = O(\log n)$. Also, as with the SUM operation, any associative function can be used in place of the $+$. In fact, the algorithm described can be used more generally to solve various recurrences, such as the first-order linear recurrences $x_i = (x_{i-1} \otimes a_i) \oplus b_i$, $0 \le i \le n$, where $\otimes$ and $\oplus$ are both associative [Kogge and Stone 1973].

Scans have proven so useful in the implementation of parallel algorithms that some parallel machines provide support for scan operations in hardware.

10.4.3 Multiprefix and Fetch-and-Add

The multiprefix operation is a generalization of the scan operation in which multiple independent scans are performed. The input to the multiprefix operation is a sequence A of n pairs (k, a), where k specifies a key and a specifies an integer data value. For each key value, the multiprefix operation performs an independent scan. The output is a sequence B of n integers containing the results of each of the scans such that if $A[i] = (k, a)$ then

$$B[i] = \text{sum}(\{b : (t, b) \in A[0..i) | t = k\})$$

In other words, each position receives the sum of all previous elements that have the same key. As an example,

$$\text{MULTIPREFIX}([(1,5),(0,2),(0,3),(1,4),(0,1),(2,2)])$$

returns the sequence

$$[0,0,2,5,5,0]$$

The *fetch-and-add* operation is a weaker version of the multiprefix operation, in which the order of the input elements for each scan is not necessarily the same as their order in the input sequence A. In this chapter we omit the implementation of the multiprefix operation, but it can be solved by a function that requires work $O(n)$ and depth $O(\log n)$ using concurrent writes [Matias and Vishkin 1991].

10.4.4 Pointer Jumping

Pointer jumping is a technique that can be applied to both linked lists and trees [Wyllie 1979]. The basic pointer jumping operation is simple. Each node i replaces its pointer $P[i]$ with the pointer of the node that it points to, $P[P[i]]$. By repeating this operation, it is possible to compute, for each node in a list or tree, a pointer to the end of the list or root of the tree. Given set P of pointers that represent a tree (i.e., pointers from children to their parents), the following code will generate a pointer from each node to the root of the tree. We assume that the root points to itself.

Algorithm: POINT_TO_ROOT(P).

for j **from** 1 **to** $\lceil \log |P| \rceil$
    $P := \{P[P[i]] : i \in [0..|P|)\}$

The idea behind this algorithm is that in each loop iteration the distance spanned by each pointer, with respect to the original tree, will double, until it points to the root. Since a tree constructed from $n = |P|$ pointers has depth at most $n - 1$, after $\lceil \log n \rceil$ iterations each pointer will point to the root. Because each iteration has constant depth and performs $\Theta(n)$ work, the algorithm has depth $\Theta(\log n)$ and work $\Theta(n \log n)$.

10.4.5 List Ranking

The problem of computing the distance from each node to the end of a linked list is called *list ranking*. Algorithm POINT_TO_ROOT can be easily modified to compute these distances, as follows.

Algorithm: LIST_RANK(P).

$V = \{$**if** $P[i] = i$ **then** 0 **else** $1 : i \in [0..|P|)\}$
for j **from** 1 **to** $\lceil \log |P| \rceil$
    $V := \{V[i] + V[P[i]] : i \in [0..|P|)\}$
    $P := \{P[P[i]] : i \in [0..|P|)\}$
return V

In this function, $V[i]$ can be thought of as the distance spanned by pointer $P[i]$ with respect to the original list. Line 1 initializes V by setting $V[i]$ to 0 if i is the last node (i.e., points to itself), and 1 otherwise. In each iteration, line 3 calculates the new length of $P[i]$. The function has depth $\Theta(\log n)$ and work $\Theta(n \log n)$.

It is worth noting that there is a simple sequential solution to the list-ranking problem that performs only $O(n)$ work: you just walk down the list, incrementing a counter at each step. The preceding parallel algorithm, which performs $\Theta(n \log n)$ work, is not **work efficient**. There are, however, a variety of work-efficient parallel solutions to this problem.

The following parallel algorithm uses the technique of random sampling to construct a pointer from each node to the end of a list of n nodes in a work-efficient fashion [Reid-Miller 1994]. The algorithm is easily generalized to solve the list-ranking problem:

1. Pick m list nodes at random and call them the *start* nodes.
2. From each start node u, follow the list until reaching the next start node v. Call the list nodes between u and v the *sublist* of u.
3. Form a shorter list consisting only of the start nodes and the final node on the list by making each start node point to the next start node on the list.
4. Using pointer jumping on the shorter list, for each start node create a pointer to the last node in the list.
5. For each start node u, distribute the pointer to the end of the list to all of the nodes in the sublist of u.

The key to analyzing the work and depth of this algorithm is to bound the length of the longest sublist. Using elementary probability theory, it is not difficult to prove that the expected length of the longest sublist is at most $O((n \log m)/m)$. The work and depth for each step of the algorithm are thus computed as follows:

1. $W(n, m) = O(m)$ and $D(n, m) = O(1)$.
2. $W(n, m) = O(n)$ and $D(n, m) = O((n \log m)/m)$.
3. $W(n, m) = O(m)$ and $D(n, m) = O(1)$.
4. $W(n, m) = O(m \log m)$ and $D(n, m) = O(\log m)$.
5. $W(n, m) = O(n)$ and $D(n, m) = O((n \log m)/m)$.

Thus, the work for the entire algorithm is $W(m, n) = O(n + m \log m)$, and the depth is $O((n \log m)/m)$. If we set $m = n/ \log n$, these reduce to $W(n) = O(n)$ and $D(n) = O(\log^2 n)$.

Using a technique called **contraction**, it is possible to design a list ranking algorithm that runs in $O(n)$ work and $O(\log n)$ depth [Anderson and Miller 1988, 1990]. This technique also can be applied to trees [Miller and Reif 1989, 1991].

10.4.6 Removing Duplicates

Given a sequence of items, the remove-duplicates algorithm removes all duplicates, returning the resulting sequence. The order of the resulting sequence does not matter.

10.4.6.1 Approach 1: Using an Array of Flags

If the items are all nonnegative integers drawn from a small range, we can use a technique similar to bucket sort to remove the duplicates. We begin by creating an array equal in size to the range and initializing all of its elements to 0. Next, using concurrent writes we set a flag in the array for each number that appears in the input list. Finally, we extract those numbers whose flags are set. This algorithm is expressed as follows.

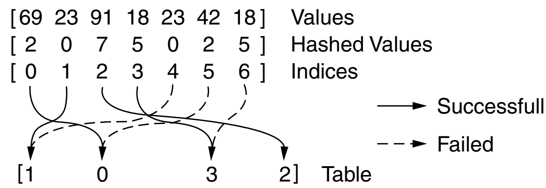


FIGURE 10.4 Each key attempts to write its index into a hash table entry.

Algorithm: REM_DUPLICATES (V).

RANGE $:= 1 + \text{MAX}(V)$
FLAGS $:= dist(0, \text{RANGE}) \leftarrow \{(i, 1) : i \in V\}$
return $\{j : j \in [0..\text{RANGE}) \mid \text{FLAGS}[j] = 1\}$

This algorithm has depth $O(1)$ and performs work $O(\text{MAX}(V))$. Its obvious disadvantage is that it explodes when given a large range of numbers, both in memory and in work.

10.4.6.2 Approach 2: Hashing

A more general approach is to use a hash table. The algorithm has the following outline. First, we create a hash table whose size is prime and approximately two times as large as the number of items in the set V. A prime size is best, because it makes designing a good hash function easier. The size also must be large enough that the chances of collisions in the hash table are not too great. Let m denote the size of the hash table. Next, we compute a hash value, $hash(V[j], m)$, for each item $V[j] \in V$ and attempt to write the index j into the hash table entry $hash(V[j], m)$. For example, Figure 10.4 describes a particular hash function applied to the sequence [69, 23, 91, 18, 23, 42, 18]. We assume that if multiple values are simultaneously written into the same memory location, one of the values will be correctly written. We call the values $V[j]$ whose indices j are successfully written into the hash table *winners*. In our example, the winners are $V[0]$, $V[1]$, $V[2]$, and $V[3]$, that is, 69, 23, 91, and 18. The winners are added to the duplicate-free set that we are constructing, and then set aside. Among the losers, we must distinguish between two types of items: those that were defeated by an item with the same value, and those that were defeated by an item with a different value. In our example, $V[5]$ and $V[6]$ (23 and 18) were defeated by items with the same value, and $V[4]$ (42) was defeated by an item with a different value. Items of the first type are set aside because they are duplicates. Items of the second type are retained, and we repeat the entire process on them using a different hash function. In general, it may take several iterations before all of the items have been set aside, and in each iteration we must use a different hash function.

Removing duplicates using hashing can be implemented as follows:

Algorithm: REMOVE_DUPLICATES (V).

$m :=$ NEXT_PRIME $(2 * |V|)$
TABLE $:= dist(-1, m)$
$i := 0$
$R := \{\}$
while $|V| > 0$
    TABLE := TABLE $\leftarrow \{(hash(V[j], m, i), j) : j \in [0..|V|)\}$
    $W := \{V[j] : j \in [0..|V|) \mid$ TABLE $[hash(V[j], m, i)] = j\}$
    $R := R \mathbin{++} W$
    TABLE := TABLE $\leftarrow \{(hash(k, m, i), k) : k \in W\}$
    $V := \{k \in V \mid$ TABLE $[hash(k, m, i)] \neq k\}$
    $i := i + 1$
return R

The first four lines of function REMOVE_DUPLICATES initialize several variables. Line 1 finds the first prime number larger than $2 * |V|$ using the built-in function NEXT_PRIME. Line 2 creates the hash table and initializes its entries with an arbitrary value (-1). Line 3 initializes i, a variable that simply counts iterations of the **while** loop. Line 4 initializes the sequence R, the result, to be empty. Ultimately, R will contain a single copy of each distinct item in the sequence V.

The bulk of the work in function REMOVE_DUPLICATES is performed by the **while** loop. Although there are items remaining to be processed, we perform the following steps. In line 6, each item $V[j]$ attempts to write its index j into the table entry given by the hash function $hash(V[j], m, i)$. Note that the hash function takes the iteration i as an argument, so that a different hash function is used in each iteration. Concurrent writes are used so that if several items attempt to write to the same entry, precisely one will win. Line 7 determines which items successfully wrote their indices in line 6 and stores their values in an array called W (for *winners*). The winners are added to the result array R in line 8. The purpose of lines 9 and 10 is to remove all of the items that are either winners or duplicates of winners. These lines reuse the hash table. In line 9, each winner writes its value, rather than its index, into the hash table. In this step there are no concurrent writes. Finally, in line 10, an item is retained only if it is not a winner, and the item that defeated it has a different value.

It is not difficult to prove that, with high probability, each iteration reduces the number of items remaining by some constant fraction until the number of items remaining is small. As a consequence, $D(n) = O(\log n)$ and $W(n) = O(n)$.

The remove-duplicates algorithm is frequently used for set operations; for instance, there is a trivial implementation of the set union operation given the code for REMOVE_DUPLICATES.

10.5 Graphs

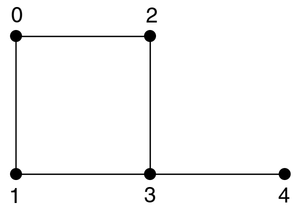
Graphs present some of the most challenging problems to parallelize since many standard sequential graph techniques, such as depth-first or priority-first search, do not parallelize well. For some problems, such as minimum spanning tree and biconnected components, new techniques have been developed to generate efficient parallel algorithms. For other problems, such as single-source shortest paths, there are no known efficient parallel algorithms, at least not for the general case.

We have already outlined some of the parallel graph techniques in Section 10.3. In this section we describe algorithms for breadth-first search, connected components, and minimum spanning trees. These algorithms use some of the general techniques. In particular, randomization and graph contraction will play an important role in the algorithms. In this chapter we will limit ourselves to algorithms on sparse undirected graphs. We suggest the following sources for further information on parallel graph algorithms Reif [1993, Chap. 2 to 8], JáJá [1992, Chap. 5], and Gibbons and Ritter [1990, Chap. 2].

10.5.1 Graphs and Their Representation

A *graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* E in which each edge connects two vertices. In a *directed graph* each edge is directed from one vertex to another, whereas in an *undirected graph* each edge is symmetric, i.e., goes in both directions. A *weighted graph* is a graph in which each edge $e \in E$ has a weight $w(e)$ associated with it. In this chapter we will use the convention that $n = |V|$ and $m = |E|$. Qualitatively, a graph is considered sparse if $m \ll n^2$ and dense otherwise. The *diameter* of a graph, denoted $D(G)$, is the maximum, over all pairs of vertices (u, v), of the minimum number of edges that must be traversed to get from u to v.

There are three standard representations of graphs used in sequential algorithms: edge lists, adjacency lists, and adjacency matrices. An *edge list* consists of a list of edges, each of which is a pair of vertices. The list directly represents the set E. An *adjacency list* is an array of lists. Each array element corresponds to one vertex and contains a linked list of the neighboring vertices, i.e., the linked list for a vertex v would contain pointers to the vertices $\{u \mid (v, u) \in E\}$). An *adjacency matrix* is an $n \times n$ array A such that A_{ij} is 1 if $(i, j) \in E$ and 0 otherwise. The adjacency matrix representation is typically used only when the graph



(a)

[(0,1), (0,2), (2,3), (3,4), (1,3), (1,0), (2,0), (3,2), (4,3), (3,1)]

(b)

[(1,2), (0,3), (0,3), (1,2,4), (3)]

(c)

FIGURE 10.5 Representations of an undirected graph: (a) a graph, G, with 5 vertices and 5 edges, (b) the edge-list representation of G, and (c) the adjacency-list representation of G. Values between square brackets are elements of an array, and values between parentheses are elements of a pair.

is dense since it requires $\Theta(n^2)$ space, as opposed to $\Theta(m)$ space for the other two representations. Each of these representations can be used to represent either directed or undirected graphs.

For parallel algorithms we use similar representations for graphs. The main change we make is to replace the linked lists with arrays. In particular, the edge list is represented as an array of edges and the adjacency list is represented as an array of arrays. Using arrays instead of lists makes it easier to process the graph in parallel. In particular, they make it easy to grab a set of elements in parallel, rather than having to follow a list. Figure 10.5 shows an example of our representations for an undirected graph. Note that for the edge-list representation of the undirected graph each edge appears twice, once in each direction. We assume these double edges for the algorithms we describe in this chapter.* To represent a directed graph we simply store the edge only once in the desired direction. In the text we will refer to the left element of an edge pair as the *source vertex* and the right element as the *destination vertex*.

In algorithms it is sometimes more efficient to use the edge list and sometimes more efficient to use an adjacency list. It is, therefore, important to be able to convert between the two representations. To convert from an adjacency list to an edge list (representation c to representation b in Fig. 10.5) is straightforward. The following code will do it with linear work and constant depth:

$$\mathit{flatten}(\{\{(i, j) : j \in G[i]\} : i \in [0 \cdots |G|\})$$

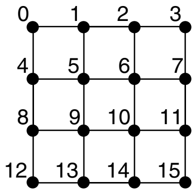
where G is the graph in the adjacency list representation. For each vertex i this code pairs up each of i's neighbors with i and then flattens the results.

To convert from an edge list to an adjacency list is somewhat more involved but still requires only linear work. The basic idea is to sort the edges based on the source vertex. This places edges from a particular vertex in consecutive positions in the resulting array. This array can then be partitioned into blocks based on the source vertices. It turns out that since the sorting is on integers in the range $[0 \ldots |V|)$, a radix sort can be used (see radix sort subsection in Section 10.6), which can be implemented in linear work. The depth of the radix sort depends on the depth of the multiprefix operation. (See previous subsection on multiprefix.)

10.5.2 Breadth-First Search

The first algorithm we consider is parallel breadth-first search (BFS). BFS can be used to solve various problems such as finding if a graph is connected or generating a spanning tree of a graph. Parallel BFS is similar to the sequential version, which starts with a source vertex s and visits levels of the graph one after the other using a queue. The main difference is that each level is going to be visited in parallel and no queue is required. As with the sequential algorithm, each vertex will be visited only once and each edge, at most twice, once in each direction. The work is therefore linear in the size of the graph $O(n + m)$. For a graph with diameter D, the number of levels processed by the algorithm will be at least $D/2$ and at most

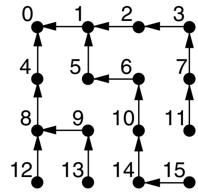
*If space is of serious concern, the algorithms can be easily modified to work with edges stored in just one direction.



(a)

Step	Frontier
0	[0]
1	[1,4]
2	[2,5,8]
3	[3,6,9,12]
5	[7,10,13]
6	[11,14]
7	[15]

(b)



(c)

FIGURE 10.6 Example of parallel breadth-first search: (a) a graph, G, (b) the frontier at each step of the BFS of G with $s = 0$, and (c) a BFS tree.

D, depending on where the search is initiated. We will show that each level can be processed in constant depth assuming a concurrent-write model, so that the total depth of parallel BFS is $O(D)$.

The main idea of parallel BFS is to maintain a set of frontier vertices, which represent the current level being visited, and to produce a new frontier on each step. The set of frontier vertices is initialized with the singleton s (the source vertex) and during the execution of the algorithm each vertex will be visited only once. A new frontier is generated by collecting all of the neighbors of the current frontier vertices in parallel and removing any that have already been visited. This is not sufficient on its own, however, since multiple vertices might collect the same unvisited vertex. For example, consider the graph in Figure 10.6. On step 2 vertices 5 and 8 will both collect vertex 9. The vertex will therefore appear twice in the new frontier. If the duplicate vertices are not removed, the algorithm can generate an exponential number of vertices in the frontier. This problem does not occur in the sequential BFS because vertices are visited one at a time. The parallel version therefore requires an extra step to remove duplicates.

The following algorithm implements the parallel BFS. It takes as input a source vertex s and a graph G represented as an adjacency array and returns as its result a breadth-first search tree of G. In a BFS tree each vertex processed at level i points to one of its neighbors processed at level $i - 1$ [see Figure 10.6c]. The source s is the root of the tree.

Algorithm: BFS (s, G).

1 $Fr := [s]$
2 $Tr := dist(-1, |G|)$
3 $Tr[s] := s$
4 **while** $(|Fr| \neq 0)$
5 $\quad E := flatten(\{\{(u, v) : u \in G[v]\} : v \in Fr\})$
6 $\quad E' := \{(u, v) \in E \mid Tr[u] = -1\}$
7 $\quad Tr := Tr \leftarrow E'$
8 $\quad Fr := \{u : (u, v) \in E' \mid v = Tr[u]\}$
9 **return** Tr

In this code Fr is the set of frontier vertices, and Tr is the current BFS tree, represented as an array of indices (pointers). The pointers in Tr are all initialized to -1, except for the source s, which is initialized to point to itself. The algorithm assumes the arbitrary concurrent-write model.

We now consider each iteration of the algorithm. The iterations terminate when there are no more vertices in the frontier (line 4). The new frontier is generated by first collecting together the set of edges from the current frontier vertices to their neighbors into an edge array (line 5). An edge from v to u is represented as the pair (u, v). We then remove any edges whose destination has already been visited (line 6). Now each edge writes its source index into the destination vertex (line 7). In the case that more than one

edge has the same destination, one of the source vertices will be written arbitrarily; this is the only place the algorithm will require a concurrent write. These indices will act as the back pointers for the BFS tree, and they also will be used to remove the duplicates for the next frontier set. In particular, each edge checks whether it succeeded by reading back from the destination, and if it succeeded, then the destination is included in the new frontier (line 8). Since only one edge that points to a given destination vertex will succeed, no duplicates will appear in the new frontier.

The algorithm requires only constant depth per iteration of the while loop. Since each vertex and its associated edges are visited only once, the total work is $O(m + n)$. An interesting aspect of this parallel BFS is that it can generate BFS trees that cannot be generated by a sequential BFS, even allowing for any order of visiting neighbors in the sequential BFS. We leave the generation of an example as an exercise. We note, however, that if the algorithm used a priority concurrent write (see previous subsection describing the model used in this chapter) on line 7, then it would generate the same tree as a sequential BFS.

10.5.3 Connected Components

We now consider the problem of labeling the connected components of an undirected graph. The problem is to label all of the vertices in a graph G such that two vertices u and v have the same label if and only if there is a path between the two vertices. Sequentially, the connected components of a graph can easily be labeled using either depth-first or breadth-first search. We have seen how to implement breadth-first search, but the technique requires a depth proportional to the diameter of a graph. This is fine for graphs with a small diameter, but it does not work well in the general case. Unfortunately, in terms of work, even the most efficient polylogarithmic depth parallel algorithms for depth-first search and breadth-first search are very inefficient. Hence, the efficient algorithms for solving the connected components problem use different techniques.

The two algorithms we consider are based on graph contraction. Graph contraction proceeds by contracting the vertices of a connected subgraph into a single vertex to form a new smaller graph. The techniques we use allow the algorithms to make many such contractions in parallel across the graph. The algorithms, therefore, proceed in a sequence of steps, each of which contracts a set of subgraphs, and forms a smaller graph in which each subgraph has been converted into a vertex. If each such step of the algorithm contracts the size of the graph by a constant fraction, then each component will contract down to a single vertex in $O(\log n)$ steps. By running the contraction in reverse, the algorithms can label all of the vertices in the components. The two algorithms we consider differ in how they select subgraphs for contraction. The first uses randomization and the second is deterministic. Neither algorithm is work efficient because they require $O((n + m) \log n)$ work for worst-case graphs, but we briefly discuss how they can be made to be work efficient in the subsequent improved version subsection. Both algorithms require the concurrent-write model.

10.5.3.1 Random Mate Graph Contraction

The random mate technique for graph contraction is based on forming a set of star subgraphs and contracting the stars. A *star* is a tree of depth one; it consists of a root and an arbitrary number of children. The random mate algorithm finds a set of nonoverlapping stars in a graph and then contracts each star into a single vertex by merging the children into their parents. The technique used to form the stars uses randomization. It works by having each vertex flip a coin and then identify itself as either a parent or a child based on the outcome. We assume the coin is unbiased so that every vertex has a 50% probability of being a parent. Now every vertex that has come up a child looks at its neighbors to see if any are parents. If at least one is a parent, then the child picks one of the neighboring parents as its parent. This process has selected a set of stars, which can be contracted. When contracting, we relabel all of the edges that were incident on a contracting child to its parent's label. Figure 10.7 illustrates a full contraction step. This contraction step is repeated until all components are of size 1.

To analyze the costs of the algorithm we need to know how many vertices are expected to be removed on each contraction step. First, we note that the step is going to remove only children and only if they have

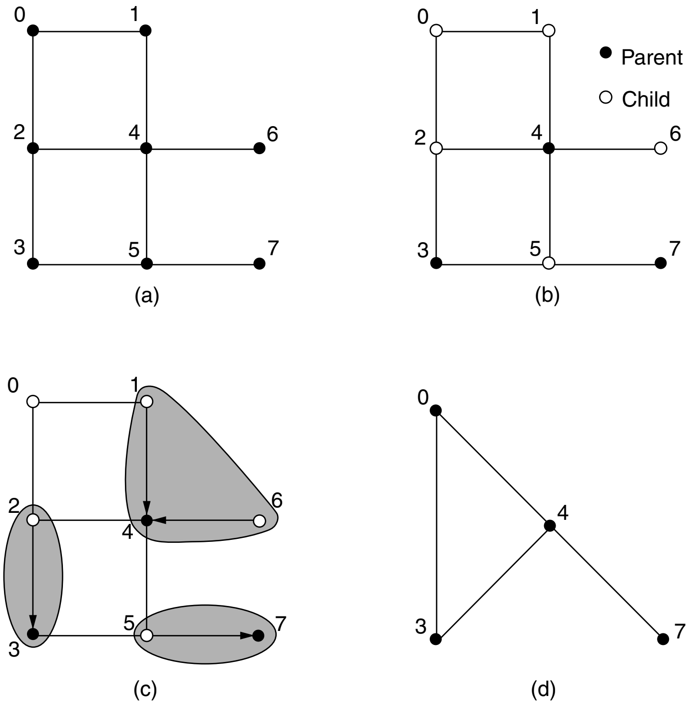


FIGURE 10.7 Example of one step of random mate graph contraction: (a) the original graph G, (b) G after selecting the parents randomly, (c) contracting the children into the parents (the shaded regions show the subgraphs), and (d) the contracted graph G'.

a neighboring parent. The probability that a vertex will be deleted is therefore the probability that it is a child multiplied by the probability that at least one of its neighbors is a parent. The probability that it is a child is 1/2 and the probability that at least one neighbor is a parent is at least 1/2 (every vertex has one or more neighbors, otherwise it would be completed). We, therefore, expect to remove at least 1/4 of the remaining vertices at each step and expect the algorithm to complete in no more than $\log_{4/3} n$ steps. The full probabilistic analysis is somewhat more involved since we could have a streak of bad flips, but it is not too hard to show that the algorithm is very unlikely to require more than $O(\log n)$ steps.

The following algorithm implements the random mate technique. The input is a graph G in the edge list representation (note that this is a different representation than used in BFS), along with the labels L of the vertices. We assume the labels are initialized to the index of the vertex. The output of the algorithm is a label for each vertex, such that all vertices in a component will be labeled with one of the original labels of a vertex in the component.

Algorithm: CC_RANDOM_MATE (L, E).

if $(|E| = 0)$ **then return** L
else
    CHILD $:= \{rand\_bit() : v \in [1..n]\}$
    $H := \{(u, v) \in E \mid \text{CHILD}[u] \wedge \neg\text{CHILD}[v]\}$
    $L := L \leftarrow H$
    $E' := \{(L[u], L[v]) : (u, v) \in E \mid L[u] \neq L[v]\}$
    $L' :=$ CC_RANDOM_MATE(L, E')
    $L' := L' \leftarrow \{(u, L'[v]) : (u, v) \in H\}$
    return L'

The algorithm works recursively by contracting the graph, labeling the components of the contracted graph, and then passing the labels to the children of the original graph. The termination condition is when there are no more edges (line 1). To make a contraction step the algorithm first flips a coin on each vertex (line 3). Now the algorithm subselects the edges-with a child on the left and a parent on the right (line 4). These are called the *hook edges*. Each of the hook edges-writes the parent index into the child's label (line 5). If a child has multiple neighboring parents, then one of the parents will be written arbitrarily; we are assuming an arbitrary concurrent write. At this point each child is labeled with one of its neighboring parents, if it has one. Now all edges update themselves to point to the parents by reading from their two endpoints and using these as their new endpoints (line 6). In the same step the edges can check if their two endpoints are within the same contracted vertex (self-edges) and remove themselves if they are. This gives a new sequence of edges E^1. The algorithm has now completed the contraction step and is called recursively on the contracted graph (line 7). The resulting labeling L' of the recursive call is used to update the labels of the children (line 8).

Two things should be noted about this algorithm. First, the algorithm flips coins on all of the vertices on each step even though many have already been contracted (there are no more edges that point to them). It turns out that this will not affect our worst-case asymptotic work or depth bounds, but in practice it is not hard to flip coins only on active vertices by keeping track of them: just keep an array of the labels of the active vertices. Second, if there are cycles in the graph, then the algorithm will create redundant edges in the contracted subgraphs. Again, keeping these edges is not a problem for the correctness or cost bounds, but they could be removed using hashing as previously discussed in the section on removing duplicates.

To analyze the full work and depth of the algorithm we note that each step requires only constant depth and $O(n+m)$ work. Since the number of steps is $O(\log n)$ with high probability, as mentioned earlier, the total depth is $O(\log n)$ and the work is $O((n+m)\log n)$, both with high probability. One might expect that the work would be linear since the algorithm reduces the number of vertices on each step by a constant fraction. We have no guarantee, however, that the number of edges also is going to contract geometrically, and in fact for certain graphs they will not. Subsequently, in this section we will discuss how this can be improved to lead to a work-efficient algorithm.

10.5.3.2 Deterministic Graph Contraction

Our second algorithm for graph contraction is deterministic [Greiner 1994]. It is based on forming trees as subgraphs and contracting these trees into a single vertex using pointer jumping. To understand the algorithm, consider the graph in Figure 10.8a. The overall goal is to contract all of the vertices of the

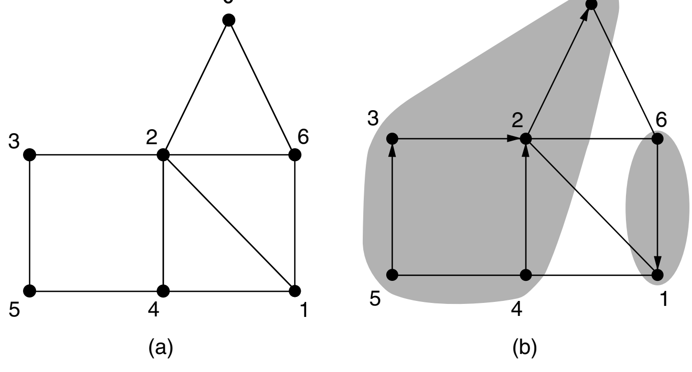


FIGURE 10.8 Tree-based graph contraction: (a) a graph, G, and (b) the hook edges induced by hooking larger to smaller vertices and the subgraphs induced by the trees.

graph into a single vertex. If we had a spanning tree that was imposed on the graph, we could contract the graph by contracting the tree using pointer jumping as discussed previously. Unfortunately, finding a spanning tree turns out to be as hard as finding the connected components of the graph. Instead, we will settle for finding a number of trees that cover the graph, contract each of these as our subgraphs using pointer jumping, and then recurse on the smaller graph. To generate the trees, the algorithm hooks each vertex into a neighbor with a smaller label. This guarantees that there are no cycles since we are only generating pointers from larger to smaller numbered vertices. This hooking will impose a set of disjoint trees on the graph. Figure 10.8b shows an example of such a hooking step. Since a vertex can have more than one neighbor with a smaller label, there can be many possible hookings for a given graph. For example, in Figure 10.8, vertex 2 could have hooked into vertex 1.

The following algorithm implements the tree-based graph contraction. We assume that the labels L are initialized to the index of the vertex.

Algorithm: CC_TREE_CONTRACT(L, E).

1 **if**$(|E| = 0)$
2 **then return** L
3 **else**
4 $\quad H := \{(u, v) \in E \mid u < v\}$
5 $\quad L := L \leftarrow H$
6 $\quad L :=$ POINT_TO_ROOT(L)
7 $\quad E' := \{(L[u], L[v]) : (u, v) \in E \mid L[u] \neq L[v]\}$
8 $\quad$**return** CC_TREE_CONTRACT(L, E')

The structure of the algorithm is similar to the random mate graph contraction algorithm. The main differences are inhow the hooks are selected (line 4), the pointer jumping step to contract the trees (line 6), and the fact that no relabeling is required when returning from the recursive call. The hooking step simply selects edges that point from smaller numbered vertices to larger numbered vertices. This is called a *conditional hook*. The pointer jumping step uses the algorithm given earlier in Section 10.4. This labels every vertex in the tree with the root of the tree. The edge relabeling is the same as in a random mate algorithm. The reason we do not need to relabel the vertices after the recursive call is that the pointer jumping will do the relabeling.

Although the basic algorithm we have described so far works well in practice, in the worst case it can take $n - 1$ steps. Consider the graph in Figure 10.9a. After hooking and contracting, only one vertex has been removed. This could be repeated up to $n - 1$ times. This worst-case behavior can be avoided by trying to hook in both directions (from larger to smaller and from smaller to larger) and picking the hooking that hooks more vertices. We will make use of the following lemma.

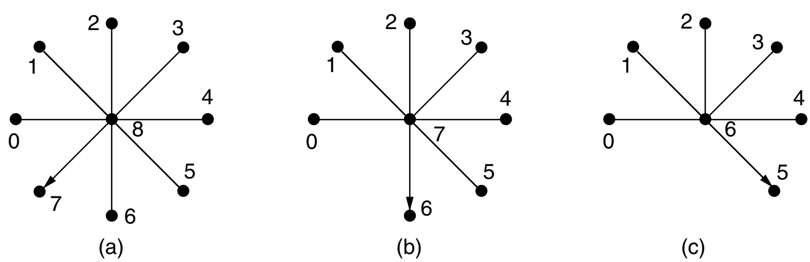


FIGURE 10.9 A worst-case graph: (a) a star graph, G, with the maximum index at the root of the star, (b) G after one step of contraction, and (c) G after two steps of contraction.

Lemma 10.1 *Let $G = (V, E)$ be an undirected graph in which each vertex has at least one neighbor, then either $|\{u|(u, v) \in E, u < v\}| \geq |V|/2$ or $|\{u|(u, v) \in E, u > v\}| > |V|/2$.*

***Proof* 10.2** Every vertex must have either a neighbor with a lesser index or a neighbor with a greater index. This means that if we consider the set of vertices with a lesser neighbor and the set of vertices with a greater neighbor, then one of those sets must consist of at least one-half the vertices. □

This lemma will guarantee that if we try hooking in both directions and pick the better one we will remove at least one-half of the vertices on each step, so that the number of steps is bounded by $\log n$.

We now consider the total cost of the algorithm. The hooking and relabeling of edges on each step takes $O(m)$ work and constant depth. The tree contraction using pointer jumping on each step requires $O(n \log n)$ work and $O(\log n)$ depth, in the worst case. Since there are $O(\log n)$ steps, in the worst case, the total work is $O((m + n \log n) \log n)$ and depth $O(\log^2 n)$. However, if we keep track of the active vertices (the roots) and only pointer jump on active vertices, then the work is reduced to $O((m + n) \log n)$ since the number of vertices geometrically decreases. This requires that the algorithm relabels on the way back up the recursion as done for the random mate algorithm. The total work with this modification is the same work as the randomized technique, although the depth has increased.

10.5.3.3 Improved Versions of Connected Components

There are many improvements to the two basic connected component algorithms we described. Here we mention some of them.

The deterministic algorithm can be improved to run in $O(\log n)$ depth with the same work bounds [Awerbuch and Shiloach 1987, Shiloach and Vishkin 1982]. The basic idea is to interleave the hooking steps with the **shortcutting** steps. The one tricky aspect is that we must always hook in the same direction (i.e., from smaller to larger), so as not to create cycles. Our previous technique to solve the star-graph problem, therefore, does not work. Instead, each vertex checks if it belongs to any tree after hooking. If it does not, then it can hook to any neighbor, even if it has a larger index. This is called an *unconditional hook*.

The randomized algorithm can be improved to run in optimal work $O(n + m)$ [Gazit 1991]. The basic idea is to not use all of the edges for hooking on each step and instead use a sample of the edges. This basic technique developed for parallel algorithms has since been used to improve some sequential algorithms, such as deriving the first linear work algorithm for minimum spanning trees [Klein and Tarjan 1994].

Another improvement is to use the EREW model instead of requiring concurrent reads and writes [Halperin and Zwick 1994]. However, this comes at the cost of greatly complicating the algorithm. The basic idea is to keep circular linked lists of the neighbors of each vertex and then to splice these lists when merging vertices.

10.5.3.4 Extensions to Spanning Trees and Minimum Spanning Trees

The connected component algorithms can be extended to finding a spanning tree of a graph or minimum spanning tree of a weighted graph. In both cases we assume the graphs are undirected.

A *spanning tree* of a connected graph $G = (V, E)$ is a connected graph $T = (V, E')$ such that $E' \subseteq E$ and $|E'| = |V| - 1$. Because of the bound on the number of edges, the graph T cannot have any cycles and therefore forms a tree. Any given graph can have many different spanning trees.

It is not hard to extend the connectivity algorithms to return the spanning tree. In particular, whenever two components are hooked together the algorithm can keep track of which edges were used for hooking. Since each edge will hook together two components that are not connected yet, and only one edge will succeed in hooking the components, the collection of these edges across all steps will form a spanning tree (they will connect all vertices and have no cycles). To determine which edges were used for contraction, each edge checks if it successfully hooked after the attempted hook.

A minimum spanning tree of a connected weighted graph $G = (V, E)$ with weights $w(e)$ for $e \in E$ is a spanning tree $T = (V, E')$ of G such that

$$w(T) = \sum_{e \in E'} w(e)$$

is minimized. The connected component algorithms also can be extended to determine the minimum spanning tree. Here we will briefly consider an extension of the random mate technique. The algorithm will take advantage of the property that, given any $W \subset V$, the minimum edge from W to $V - W$ must be in some minimum spanning tree. This implies that the minimum edge incident on a vertex will be on a minimum spanning tree. This will be true even after we contract subgraphs into vertices since each subgraph is a subset of V.

To implement the minimum spanning tree algorithm we therefore modify the random mate technique so that each child u, instead of picking an arbitrary parent to hook into, finds the incident edge (u, v) with minimum weight and hooks into v if it is a parent. If v is not a parent, then the child u does nothing (it is left as an orphan). Figure 10.10 illustrates the algorithm. As with the spanning tree algorithm, we keep track of the edges we use for hooks and add them to a set E'. This new rule will still remove 1/4 of the vertices on each step on average since a vertex has 1/2 probability of being a child, and there is 1/2 probability that the vertex at the other end of the minimum edge is a parent. The one complication in this minimum spanning tree algorithm is finding for each child the incident edge with minimum weight. Since we are keeping an edge list, this is not trivial to compute. If we had an adjacency list, then it would be easy, but since we are updating the endpoints of the edges, it is not easy to maintain the adjacency list. One way to solve this problem is to use a priority concurrent write. In such a write, if multiple values are written to the same location, the one coming from the leftmost position will be written. With such a scheme the minimum edge can be found by presorting the edges by their weight so that the lowest weighted edge will always win when executing a concurrent write. Assuming a priority write, this minimum spanning tree algorithm has the same work and depth as the random mate connected components algorithm.

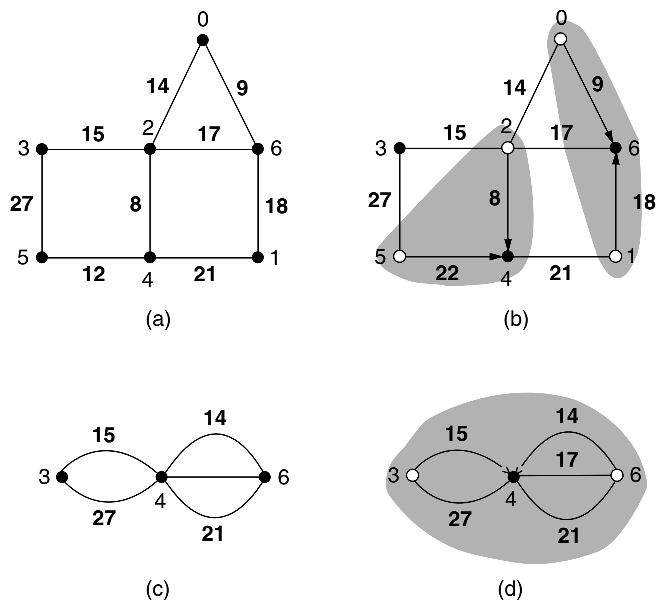


FIGURE 10.10 Example of the minimum spanning tree algorithm. (a) The original weighted graph G. (b) Each child (light) hooks across its minimum weighted edge to a parent (dark), if the edge is incident on a parent. (c) The graph after one step of contraction. (d) The second step in which children hook across minimum weighted edges to parents.

10.6 Sorting

Sorting is a problem that admits a variety of parallel solutions. In this section we limit our discussion to two parallel sorting algorithms, QuickSort and radix sort. Both of these algorithms are easy to program, and both work well in practice. Many more sorting algorithms can be found in the literature. The interested reader is referred to Akl [1985], JáJá [1992], and Leighton [1992] for more complete coverage.

10.6.1 QuickSort

We begin our discussion of sorting with a parallel version of QuickSort. This algorithm is one of the simplest to code.

Algorithm: QUICKSORT(A).

if $|A| = 1$ **then return** A
$i := rand\_int(|A|)$
$p := A[i]$
in parallel do
  $L :=$ QUICKSORT($\{a : a \in A \mid a < p\}$)
  $E := \{a : a \in A \mid a = p\}$
  $G :=$ QUICKSORT($\{a : a \in A \mid a > p\}$)
return L ++ E ++ G

We can make an optimistic estimate of the work and depth of this algorithm by assuming that each time a partition element, p, is selected, it divides the set A so that neither L nor H has more than half of the elements. In this case, the work and depth are given by the recurrences

$$W(n) = 2W(n/2) + O(n)$$
$$D(n) = D(n/2) + 1$$

whose solutions are $W(n) = O(n \log n)$ and $D(n) = O(\log n)$. A more sophisticated analysis [Knuth 1973] shows that the expected work and depth are indeed $W(n) = O(n \log n)$ and $D(n) = O(\log n)$, independent of the values in the input sequence A.

In practice, the performance of parallel QuickSort can be improved by selecting more than one partition element. In particular, on a machine with P processors, choosing $P - 1$ partition elements divides the keys into P sets, each of which can be sorted by a different processor using a fast sequential sorting algorithm. Since the algorithm does not finish until the last processor finishes, it is important to assign approximately the same number of keys to each processor. Simply choosing $p - 1$ partition elements at random is unlikely to yield a good partition. The partition can be improved, however, by choosing a larger number, *sp*, of candidate partition elements at random, sorting the candidates (perhaps using some other sorting algorithm), and then choosing the candidates with ranks $s, 2s, \ldots, (p - 1)s$ to be the partition elements. The ratio s of candidates to partition elements is called the *oversampling ratio*. As s increases, the quality of the partition increases, but so does the time to sort the *sp* candidates. Hence, there is an optimum value of s, typically larger than one, which minimizes the total time. The sorting algorithm that selects partition elements in this fashion is called *sample sort* [Blelloch et al. 1991, Huang and Chow 1983, Reif and Valiant 1983].

10.6.2 Radix Sort

Our next sorting algorithm is radix sort, an algorithm that performs well in practice. Unlike QuickSort, radix sort is not a *comparison sort*, meaning that it does not compare keys directly in order to determine the relative ordering of keys. Instead, it relies on the representation of keys as b-bit integers.

The basic radix sort algorithm (whether serial or parallel) examines the keys to be sorted one *digit* at a time, starting with the least significant digit in each key. Of fundamental importance is that this intermediate sort on digits be *stable*: the output ordering must preserve the input order of any two keys whose bits are the same.

The most common implementation of the intermediate sort is as a counting sort. A counting sort first counts to determine the *rank* of each key — its position in the output order — and then we permute the keys to their respective locations. The following algorithm implements radix sort assuming one-bit digits.

Algorithm: RADIX_SORT(A, b)

1 **for** i **from** 0 **to** $b - 1$
2 $\quad B := \{(a \gg i) \; mod \, 2 \; : \; a \in A\}$
3 $\quad NB := \{1 - b \; : \; b \in B\}$
4 $\quad R_0 :=$ SCAN(NB)
5 $\quad s_0 :=$ SUM(NB)
6 $\quad R_1 :=$ SCAN(B)
7 $\quad R :=$ {**if** $B[j] = 0$ **then** $R_0[j]$ **else** $R_1[j] + s_0 : j \in [0..|A|)$}
8 $\quad A := A \leftarrow \{(R[j], A[j]) : j \in [0..|A|)\}$
9 **return** A

For keys with b bits, the algorithm consists of b sequential iterations of a **for** loop, each iteration sorting according to one of the bits. Lines 2 and 3 compute the value and inverse value of the bit in the current position for each key. The notation $a \gg i$ denotes the operation of shifting a i bit positions to the right. Line 4 computes the rank of each key whose bit value is 0. Computing the ranks of the keys with bit value 1 is a little more complicated, since these keys follow the keys with bit value 0. Line 5 computes the number of keys with bit value 0, which serves as the rank of the first key whose bit value is 1. Line 6 computes the relative order of the keys with bit value 1. Line 7 merges the ranks of the even keys with those of the odd keys. Finally, line 8 permutes the keys according to their ranks.

The work and depth of RADIX_SORT are computed as follows. There are b iterations of the **for** loop. In each iteration, the depths of lines 2, 3, 7, 8, and 9 are constant, and the depths of lines 4, 5, and 6 are $O(\log n)$. Hence, the depth of the algorithm is $O(b \log n)$. The work performed by each of lines 2–9 is $O(n)$. Hence, the work of the algorithm is $O(bn)$.

The radix sort algorithm can be generalized so that each b-bit key is viewed as b/r blocks of r bits each, rather than as b individual bits. In the generalized algorithm, there are b/r iterations of the **for** loop, each of which invokes the SCAN function 2^r times. When r is large, a multiprefix operation can be used for generating the ranks instead of executing a SCAN for each possible value [Blelloch et al. 1991]. In this case, and assuming the multiprefix runs in linear work, it is not hard to show that as long as $b = O(\log n)$, the total work for the radix sort is $O(n)$, and the depth is the same order as the depth of the multiprefix.

Floating-point numbers also can be sorted using radix sort. With a few simple bit manipulations, floating-point keys can be converted to integer keys with the same ordering and key size. For example, IEEE double-precision floating-point numbers can be sorted by inverting the mantissa and exponent bits if the sign bit is 1 and then inverting the sign bit. The keys are then sorted as if they were integers.

10.7 Computational Geometry

Problems in computational geometry involve determining various properties about sets of objects in a k-dimensional space. Some standard problems include finding the closest distance between a pair of points (closest pair), finding the smallest convex region that encloses a set of points (convex hull), and finding line or polygon intersections. Efficient parallel algorithms have been developed for most standard problems in computational geometry. Many of the sequential algorithms are based on divide-and-conquer and lead in a relatively straightforward manner to efficient parallel algorithms. Some others are based on a technique called plane sweeping, which does not parallelize well, but for which an analogous parallel technique, the

plane sweep tree has been developed [Aggarwal et al. 1988, Atallah et al. 1989]. In this section we describe parallel algorithms for two problems in two dimensions — closest pair and convex hull. For the convex hull we describe two algorithms. These algorithms are good examples of how sequential algorithms can be parallelized in a straightforward manner.

We suggest the following sources for further information on parallel algorithms for computational geometry: Reif [1993, Chap. 9 and Chap. 11], JáJá [1992, Chap. 6], and Goodrich [1996].

10.7.1 Closest Pair

The *closest pair problem* takes a set of points in k dimensions and returns the two points that are closest to each other. The distance is usually defined as Euclidean distance. Here we describe a closest pair algorithm for two-dimensional space, also called the planar closest pair problem. The algorithm is a parallel version of a standard sequential algorithm [Bentley and Shamos 1976], and, for n points, it requires the same work as the sequential versions $O(n \log n)$ and has depth $O(\log^2 n)$. The work is optimal.

The algorithm uses divide-and-conquer based on splitting the points along lines parallel to the y axis and is implemented as follows.

Algorithm: CLOSEST_PAIR(P).

if $(|P| < 2)$ **then return** (P, ∞)
$x_m :=$ MEDIAN $(\{x : (x, y) \in P\})$
$L := \{(x, y) \in P \mid x < x_m\}$
$R := \{(x, y) \in P \mid x \geq x_m\}$
in parallel do
    $(L', \delta_L) :=$ CLOSEST_PAIR(L)
    $(R', \delta_R) :=$ CLOSEST_PAIR(R)
$P' :=$ MERGE_BY_Y(L', R')
$\delta_P :=$ BOUNDARY_MERGE($P', \delta_L, \delta_R, x_m$)
return (P', δ_P)

This function takes a set of points P in the plane and returns both the original points sorted along the y axis and the distance between the closest two points. The sorted points are needed to help merge the results from recursive calls and can be thrown away at the end. It would be easy to modify the routine to return the closest pair of points in addition to the distance between them. The function works by dividing the points in half based on the median x value, recursively solving the problem on each half, and then merging the results. The MERGE_BY_Y function merges L' and R' along the y axis and can use a standard parallel merge routine. The interesting aspect of the code is the BOUNDARY_MERGE routine, which works on the same principle as described by Bentley and Shamos [1976] and can be computed with $O(\log n)$ depth and $O(n)$ work. We first review the principle and then show how it is implemented in parallel.

The inputs to BOUNDARY_MERGE are the original points P sorted along the y axis, the closest distance within L and R, and the median point x_m. The closest distance in P must be either the distance δ_L, the distance δ_R, or the distance between a point in L and a point in R. For this distance to be less than δ_L or δ_R, the two points must lie within $\delta = \min(\delta_L, \delta_R)$ of the line $x = x_m$. Thus, the two vertical lines at $x_r = x_m + \delta$ and $x_l = x_m - \delta$ define the borders of a region M in which the points must lie (see Figure 10.11). If we could find the closest distance in M, call it δ_M, then the closest overall distance is $\delta_P = \min(\delta_L, \delta_R, \delta_M)$.

To find δ_M, we take advantage of the fact that not many points can be packed closely together within M since all points within L or R must be separated by at least δ. Figure 10.11 shows the tightest possible packing of points in a $2\delta \times \delta$ rectangle within M. This packing implies that if the points in M are sorted along the y axis, each point can determine the minimum distance to another point in M by looking at a fixed number of neighbors in the sorted order, at most seven in each direction. To see this, consider one of the points along the top of the $2\delta \times \delta$ rectangle. To find if there are any points below it that are closer

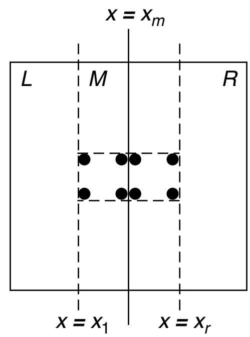


FIGURE 10.11 Merging two rectangles to determine the closest pair. Only 8 points can fit in the $2\delta \times \delta$ dashed rectangle.

than δ, it needs only to consider the points within the rectangle (points below the rectangle must be farther than δ away). As the figure illustrates, there can be at most seven other points within the rectangle. Given this property, the following function implements the border merge.

Algorithm: BOUNDARY_MERGE$(P, \delta_L, \delta_R, x_m)$.

$\delta := \min(\delta_L, \delta_R)$
$M := \{(x, y) \in P \mid (x \geq x_m - \delta) \wedge (x \leq x_m + \delta)\}$
$\delta_M := \min(\{= \min(\{\mathit{distance}(M[i], M[i+j]) : j \in [1..7]\})$
$\quad\quad\quad : i \in [0..|P-7)\}$
return $\min(\delta, \delta_M)$

In this function each point in M looks at seven points in front of it in the sorted order and determines the distance to each of these points. The minimum over all distances is taken. Since the distance relationship is symmetric, there is no need for each point to consider points behind it in the sorted order.

The work of BOUNDARY_MERGE is $O(n)$ and the depth is dominated by taking the minimum, which has $O(\log n)$ depth.* The work of the merge and median steps in CLOSEST_PAIR is also $O(n)$, and the depth of both is bounded by $O(\log n)$. The total work and depth of the algorithm therefore can be solved with the recurrences

$$W(n) = 2W(n/2) + O(n) \quad = O(n \log n)$$
$$D(n) = D(n/2) + O(\log n) = O(\log^2 n)$$

10.7.2 Planar Convex Hull

The convex hull problem takes a set of points in k dimensions and returns the smallest convex region that contains all of the points. In two dimensions, the problem is called the planar convex hull problem and it returns the set of points that form the corners of the region. These points are a subset of the original points. We will describe two parallel algorithms for the planar convex hull problem. They are both based on divide-and-conquer, but one does most of the work before the divide step, and the other does most of the work after.

*The depth of finding the minimum or maximum of a set of numbers actually can be improved to $O(\log \log n)$ with concurrent reads [Shiloach and Vishkin 1981].

10.7.2.1 QuickHull

The parallel *QuickHull* algorithm [Blelloch and Little 1994] is based on the sequential version [Preparata and Shamos 1985], so named because of its similarity to the QuickSort algorithm. As with QuickSort, the strategy is to pick a *pivot* element, split the data based on the pivot, and recurse on each of the split sets. Also as with QuickSort, the pivot element is not guaranteed to split the data into equally sized sets, and in the worst case the algorithm requires $O(n^2)$ work; however, in practice the algorithm is often very efficient, probably the most practical of the convex hull algorithms. At the end of the section we briefly describe how the splits can be made precisely so the work is bounded by $O(n \log n)$.

The QuickHull algorithm is based on the recursive function SUBHULL, which is implemented as follows.

Algorithm: SUBHULL(P, p_1, p_2).

1 $P' := \{p \in P \mid$ RIGHT_OF ?$(p, (p_1, p_2))\}$
2 **if** $(|P'| < 2)$
3 **then return** $[p_1]$ ++ P'
4 **else**
5 $\quad i :=$ MAX_INDEX($\{$DISTANCE$(p, (p_1, p_2)) : p \in P'\}$)
6 $\quad p_m := P'[i]$
7 $\quad$ **in parallel do**
8 $\quad\quad H_l :=$ SUBHULL(P', p_1, p_m)
9 $\quad\quad H_r :=$ SUBHULL(P', p_m, p_2)
10 $\quad$ **return** H_l ++ H_r

This function takes a set of points P in the plane and two points p_1 and p_2 that are known to lie on the convex hull and returns all of the points that lie on the hull clockwise from p_1 to p_2, inclusive of p_1, but not of p_2. For example, in Figure 10.12 SUBHULL$([A, B, C, \ldots, P], A, P)$ would return the sequence $[A, B, J, O]$.

The function SUBHULL works as follows. Line 1 removes all of the elements that cannot be on the hull because they lie to the right of the line from p_1 to p_2. This can easily be calculated using a cross product. If the remaining set P' is either empty or has just one element, the algorithm is done. Otherwise, the algorithm finds the point p_m farthest from the line (p_1, p_2). The point p_m must be on the hull since as a line at infinity parallel to (p_1, p_2) moves toward (p_1, p_2), it must first hit p_m. In line 5, the function MAX_INDEX returns the index of the maximum value of a sequence, using $O(n)$ work $O(\log n)$ depth, which is then used to extract the point p_m. Once p_m is found, SUBHULL is called twice recursively to find

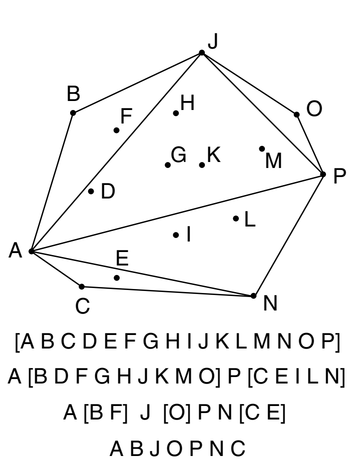


FIGURE 10.12 An example of the QuickHull algorithm.

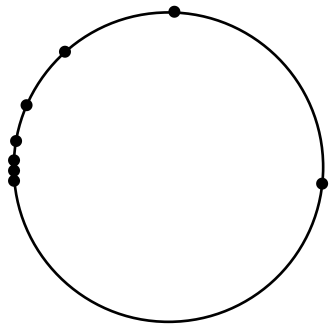


FIGURE 10.13 Contrived set of points for worst-case QuickHull.

the hulls from p_1 to p_m and from p_m to p_2. When the recursive calls return, the results are appended. The algorithm function uses SUBHULL to find the full convex hull.

Algorithm: QUICK_HULL(P).

$X := \{x : (x, y) \in P\}$
$x_{\min} := P[\mathit{min\_index}(X)]$
$x_{\max} := P[\mathit{max\_index}(X)]$
return SUBHULL($P, x_{\min}, x_{\max}$) ++ SUBHULL($P, x_{\max}, x_{\min}$)

We now consider the costs of the parallel QuickHull. The cost of everything other than the recursive calls is $O(n)$ work and $O(\log n)$ depth. If the recursive calls are balanced so that neither recursive call gets much more than half the data, then the number of levels of recursion will be $O(\log n)$. This will lead to the algorithm running in $O(\log^2 n)$ depth. Since the sum of the sizes of the recursive calls can be less than n (e.g., the points within the triangle AJP will be thrown out when making the recursive calls to find the hulls between A and J and between J and P), the work can be as little as $O(n)$ and often is in practice. As with QuickSort, however, when the recursive calls are badly partitioned, the number of levels of recursion can be as bad as $O(n)$ with work $O(n^2)$. For example, consider the case when all of the points lie on a circle and have the following unlikely distribution: $x_{\min}$ and $x_{\max}$ appear on opposite sides of the circle. There is one point that appears halfway between $x_{\min}$ and $x_{\max}$ on the sphere and this point becomes the new $x_{\max}$. The remaining points are defined recursively. That is, the points become arbitrarily close to $x_{\min}$ (see Figure 10.13). Kirkpatrick and Seidel [1986] have shown that it is possible to modify QuickHull so that it makes provably good partitions. Although the technique is shown for a sequential algorithm, it is easy to parallelize. A simplification of the technique is given by Chan et al. [1995]. This parallelizes even better and leads to an $O(\log^2 n)$ depth algorithm with $O(n \log h)$ work where h is the number of points on the convex hull.

10.7.2.2 MergeHull

The MergeHull algorithm [Overmars and Van Leeuwen 1981] is another divide-and-conquer algorithm for solving the planar convex hull problem. Unlike QuickHull, however, it does most of its work after returning from the recursive calls. The algorithm is implemented as follows.

Algorithm: MERGEHULL(P).

if ($|P| < 3$) **then return** P
else
  in parallel do
    $H_1 =$ MERGEHULL ($P[0..|P|/2)$)
    $H_2 =$ MERGEHULL ($P[|P|/2..|P|)$)
  return JOIN_HULLS(H_1, H_2)

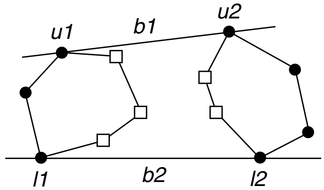


FIGURE 10.14 Merging two convex hulls.

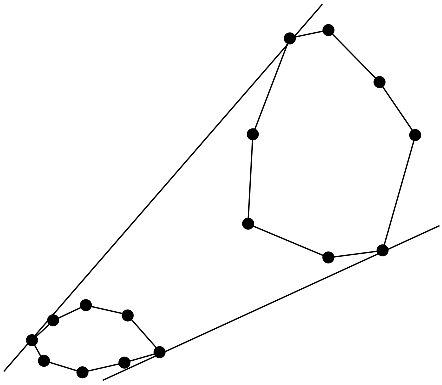


FIGURE 10.15 A bridge that is far from the top of the convex hull.

This function assumes the input P is presorted according to the x coordinates of the points. Since the points are presorted, H_1 is a convex hull on the left and H_2 is a convex hull on the right. The JOIN_HULLS routine is the interesting part of the algorithm. It takes the two hulls and merges them into one. To do this, it needs to find upper and lower points u_1 and l_1 on H_1 and u_2 and l_2 on H_2 such that u_1, u_2 and l_1, l_2 are successive points on H (see Figure 10.14). The lines b_1 and b_2 joining these upper and lower points are called the upper and lower bridges, respectively. All of the points between u_1 and l_1 and between u_2 and l_2 on the *outer* sides of H_1 and H_2 are on the final convex hull, whereas the points on the *inner* sides are not on the convex hull. Without loss of generality we consider only how to find the upper bridge b_1. Finding the lower bridge b_2 is analogous.

To find the upper bridge, one might consider taking the points with the maximum y. However, this does not work in general; u_1 can lie as far down as the point with the minimum x or maximum x value (see Figure 10.15). Instead, there is a nice solution based on binary search. Assume that the points on the convex hulls are given in order (e.g., clockwise). At each step the search algorithm will eliminate half the remaining points from consideration in either H_1 or H_2 or both. After at most $\log |H_1| + \log |H_2|$ steps the search will be left with only one point in each hull, and these will be the desired points u_1 and u_2. Figure 10.16 illustrates the rules for eliminating part of H_1 or H_2 on each step.

We now consider the cost of the algorithm. Each step of the binary search requires only constant work and depth since we only need to consider the middle two points M_1 and M_2, which can be found in constant time if the hull is kept sorted. The cost of the full binary search to find the upper bridge is therefore bounded by $D(n) = W(n) = O(\log n)$. Once we have found the upper and lower bridges, we need to remove the points on H_1 and H_2 that are not on H and append the remaining convex hull points. This requires linear work and constant depth. The overall costs of MERGEHULL are, therefore,

$$D(n) = D(n/2) + \log n = O(\log^2 n)$$
$$W(n) = 2W(n/2) + \log n + n = O(n \log n)$$

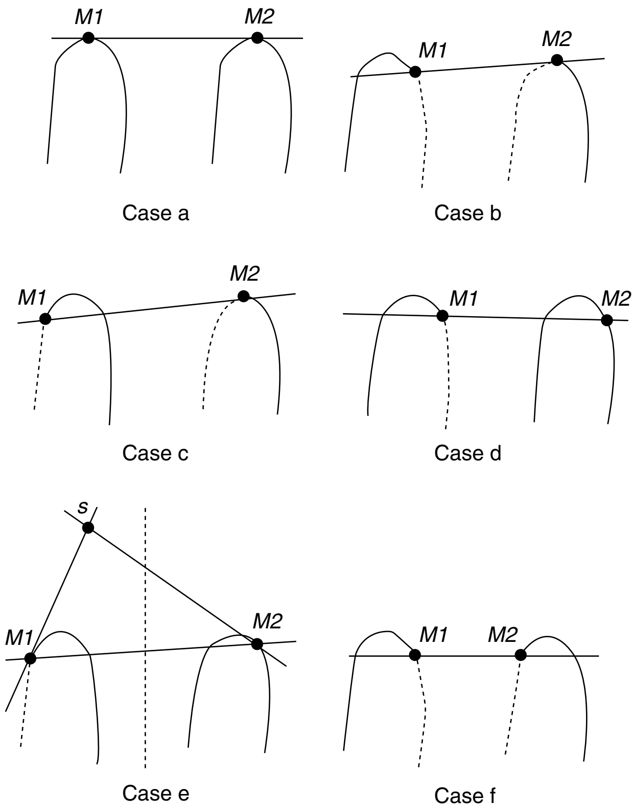


FIGURE 10.16 Cases used in the binary search for finding the upper bridge for the MergeHull. The points $M1$ and $M2$ mark the middle of the remaining hulls. The dotted lines represent the part of the hull that can be eliminated from consideration. The mirror images of cases b–e are also used. In case e, the region to eliminate depends on which side of the separating line the intersection of the tangents appears.

This algorithm can be improved to run in $O(\log n)$ depth using one of two techniques. The first involves implementing the search for the bridge points such that it runs in constant depth with linear work [Atallah and Goodrich 1988]. This involves sampling every $\sqrt{n}$th point on each hull and comparing all pairs of these two samples to narrow the search region down to regions of size $\sqrt{n}$ in constant depth. The patches then can be finished in constant depth by comparing all pairs between the two patches. The second technique [Aggarwal et al. 1988, Atallah and Goodrich 1986] uses a divide-and-conquer to separate the point set into $\sqrt{n}$ regions, solves the convex hull on each region recursively, and then merges all pairs of these regions using the binary search method. Since there are $\sqrt{n}$ regions and each of the searches takes $O(\log n)$ work, the total work for merging is $O((\sqrt{n})^2 \log n) = O(n \log n)$ and the depth is $O(\log n)$. This leads to an overall algorithm that runs in $O(n \log n)$ work and $O(\log n)$ depth.

10.8 Numerical Algorithms

There has been an immense amount of work on parallel algorithms for numerical problems. Here we briefly discuss some of the problems and results. We suggest the following sources for further information on parallel numerical algorithms: Reif [1993, Chap. 12 and Chapter 14], JáJá [1992, Chap. 8], Kumar et al. [1994, Chap. 5, Chapter 10 and Chapter 11], and Bertsekas and Tsitsiklis [1989].

10.8.1 Matrix Operations

Matrix operations form the core of many numerical algorithms and led to some of the earliest work on parallel algorithms. The most basic matrix operation is matrix multiply. The standard triply nested loop for multiplying two dense matrices is highly parallel since each of the loops can be parallelized:

Algorithm: MATRIX_MULTIPLY (A, B).

```
(l, m) := dimensions(A)
(m, n) := dimensions(B)
in parallel for i ∈ [0..l) do
    in parallel for j ∈ [0..n) do
        R_ij := sum({A_ik * B_kj : k ∈ [0..m)})
return R
```

If $l = m = n$, this routine does $O(n^3)$ work and has depth $O(\log(n))$, due to the depth of the summation. This has much more parallelism than is typically needed, and most of the research on parallel matrix multiplication has concentrated on how to use a subset of the parallelism to minimize communication costs. Sequentially, it is known that matrix multiplication can be done in better than $O(n^3)$ work. For example, Strassen's [1969] algorithm requires only $O(n^{2.81})$ work. Most of these more efficient algorithms are also easy to parallelize because of their recursive nature (Strassen's algorithm has $O(\log n)$ depth using a simple parallelization).

Another basic matrix operation is to invert matrices. Inverting dense matrices turns out to be somewhat less parallel than matrix multiplication, but still supplies plenty of parallelism for most practical purposes. When using Gauss–Jordan elimination, two of the three nested loops can be parallelized leading to an algorithm that runs with $O(n^3)$ work and $O(n)$ depth. A recursive block-based method using matrix multiplies leads to the same depth, although the work can be reduced by using one of the more efficient matrix multiplies.

Parallel algorithms for many other matrix operations have been studied, and there has also been significant work on algorithms for various special forms of matrices, such as tridiagonal, triangular, and general sparse matrices. Iterative methods for solving sparse linear systems have been an area of significant activity.

10.8.2 Fourier Transform

Another problem for which there has been a long history of parallel algorithms is the discrete Fourier transform (DFT). The fast Fourier transform (FFT) algorithm for solving the DFT is quite easy to parallelize and, as with matrix multiplication, much of the research has gone into reducing communication costs. In fact, the butterfly network topology is sometimes called the FFT network since the FFT has the same communication pattern as the network [Leighton 1992, Section 3.7]. A parallel FFT over complex numbers can be expressed as follows.

Algorithm: FFT(A).

```
n := |A|
if (n = 1) then return A
else
    in parallel do
        E := FFT({A[2i] : i ∈ [0..n/2)})
        O := FFT ({A[2i + 1] : i ∈ [0..n/2)})
    return {E[j] + O[j]e^(2πij/n) : j ∈ [0..n/2)} ++ {E[j] − O[j]e^(2πij/n) : j ∈ [0..n/2)}
```

It simply calls itself recursively on the odd and even elements and then puts the results together. This algorithm does $O(n \log n)$ work, as does the sequential version, and has a depth of $O(\log n)$.

10.9 Parallel Complexity Theory

Researchers have developed a complexity theory for parallel computation that is in some ways analogous to the theory of NP-completeness. A problem is said to belong to the class NC (Nick's class) if it can be solved in depth polylogarithmic in the size of the problem using work that is polynomial in the size of the problem [Cook 1981, Pippenger 1979]. The class NC in parallel complexity theory plays the role of P in sequential complexity, i.e., the problems in NC are thought to be tractable in parallel. Examples of problems in NC include sorting, finding minimum cost spanning trees, and finding convex hulls. A problem is said to be P-complete if it can be solved in polynomial time and if its inclusion in NC would imply that $NC = P$. Hence, the notion of P-completeness plays the role of NP-completeness in sequential complexity. (And few believe that $NC = P$.)

Although much early work in parallel algorithms aimed at showing that certain problems belong to the class NC (without considering the issue of efficiency), this work tapered off as the importance of work efficiency became evident. Also, even if a problem is P-complete, there may be efficient (but not necessarily polylogarithmic time) parallel algorithms for solving it. For example, several efficient and highly parallel algorithms are known for solving the maximum flow problem, which is P-complete.

We conclude with a short list of P-complete problems. Full definitions of these problems and proofs that they are P-complete can be found in textbooks and surveys such as Gibbons and Rytter [1990], JáJá [1992], and Karp and Ramachandran [1990]. P-complete problems are:

1. **Lexicographically first maximal independent set and clique.** Given a graph G with vertices $V = 1, 2, \ldots, n$, and a subset $S \subseteq V$, determine if S is the lexicographically first maximal independent set (or maximal clique) of G.
2. **Ordered depth-first search.** Given a graph $G = (V, E)$, an ordering of the edges at each vertex, and a subset $T \subset E$, determine if T is the depth-first search tree that the sequential depth-first algorithm would construct using this ordering of the edges.
3. **Maximum flow.**
4. **Linear programming.**
5. **The circuit value problem.** Given a Boolean circuit, and a set of inputs to the circuit, determine if the output value of the circuit is one.
6. **The binary operator generability problem.** Given a set S, an element e not in S, and a binary operator$\cdot$, determine if e can be generated from S using$\cdot$.
7. **The context-free grammar emptiness problem.** Given a context-free grammar, determine if it can generate the empty string.

Defining Terms

CRCW: This refers to a shared memory model that allows for concurrent reads (CR) and concurrent writes (CW) to the memory.

CREW: This refers to a shared memory model that allows for concurrent reads (CR) but only exclusive writes (EW) to the memory.

Depth: The longest chain of sequential dependences in a computation.

EREW: This refers to a shared memory model that allows for only exclusive reads (ER) and exclusive writes (EW) to the memory.

Graph contraction: Contracting a graph by removing a subset of the vertices.

List contraction: Contracting a list by removing a subset of the nodes.

Multiprefix: A generalization of the scan (prefix sums) operation in which the partial sums are grouped by keys.

Multiprocessor model: A model of parallel computation based on a set of communicating sequential processors.

Pipelined divide-and-conquer: A divide-and-conquer paradigm in which partial results from recursive calls can be used before the calls complete. The technique is often useful for reducing the depth of various algorithms.

Pointer jumping: In a linked structure replacing a pointer with the pointer it points to. Used for various algorithms on lists and trees. Also called recursive doubling.

PRAM model: A multiprocessor model in which all of the processors can access a shared memory for reading or writing with uniform cost.

Prefix sums: A parallel operation in which each element in an array or linked list receives the sum of all of the previous elements.

Random sampling: Using a randomly selected sample of the data to help solve a problem on the whole data.

Recursive doubling: Same as pointer jumping.

Scan: A parallel operation in which each element in an array receives the sum of all of the previous elements.

Shortcutting: Same as pointer jumping.

Symmetry breaking: A technique to break the symmetry in a structure such as a graph which can locally look the same to all of the vertices. Usually implemented with randomization.

Tree contraction: Contracting a tree by removing a subset of the nodes.

Work: The total number of operations taken by a computation.

Work-depth model: A model of parallel computation in which one keeps track of the total work and depth of a computation without worrying about how it maps onto a machine.

Work efficient: When an algorithm does no more work than some other algorithm or model. Often used when relating a parallel algorithm to the best known sequential algorithm but also used when discussing emulations of one model on another.

References

Aggarwal, A., Chazelle, B., Guibas, L., Ò'Dùnlaing, C., and Yap, C. 1988. Parallel computational geometry. *Algorithmica* 3(3):293–327.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, MA.

Akl, S. G. 1985. *Parallel Sorting Algorithms*. Academic Press, Toronto, Canada.

Anderson, R. J. and Miller, G. L. 1988. Deterministic parallel list ranking. In *Aegean Workshop on Computing: VLSI Algorithms and Architectures*. J. Reif, ed. Vol. 319, Lecture notes in computer science, pp. 81–90. Springer–Verlag, New York.

Anderson, G. L. and Miller, G. L. 1990. A simple randomized parallel algorithm for list-ranking. *Inf. Process. Lett.* 33(5):269–273.

Atallah, M. J., Cole, R., and Goodrich, M. T. 1989. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM J. Comput.* 18(3):499–532.

Atallah, M. J. and Goodrich, M. T. 1986. Efficient parallel solutions to some geometric problems. *J. Parallel Distrib. Comput.* 3(4):492–507.

Atallah, M. J. and Goodrich, M. T. 1988. Parallel algorithms for some functions of two convex polygons. *Algorithmica* 3(4):535–548.

Awerbuch, B. and Shiloach, Y. 1987. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Comput.* C-36(10):1258–1263.

Bar-Noy, A. and Kipnis, S. 1992. Designing broadcasting algorithms in the postal model for message-passing systems, pp. 13–22. In *Proc. 4th Annu. ACM Symp. Parallel Algorithms Architectures*. ACM Press, New York.

Beneš, V. E. 1965. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York.

Bentley, J. L. and Shamos, M. 1976. Divide-and-conquer in multidimensional space, pp. 220–230. In *Proc. ACM Symp. Theory Comput.* ACM Press, New York.

Bertsekas, D. P. and Tsitsiklis, J. N. 1989. *Parallel and Distributed Computation: Numerical Methods*. Prentice–Hall, Englewood Cliffs, NJ.

Blelloch, G. E. 1990. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA.

Blelloch, G. E. 1996. Programming parallel algorithms. *Commun. ACM* 39(3):85–97.

Blelloch, G. E., Chandy, K. M., and Jagannathan, S., eds. 1994. *Specification of Parallel Algorithms.* Vol. 18, DIMACS series in discrete mathematics and theoretical computer science. American Math. Soc. Providence, RI.

Blelloch, G. E. and Greiner, J. 1995. Parallelism in sequential functional languages, pp. 226–237. In *Proc. ACM Symp. Functional Programming Comput. Architecture*. ACM Press, New York.

Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. 1991. A comparison of sorting algorithms for the connection machine CM-2, pp. 3–16. In *Proc. ACM Symp. Parallel Algorithms Architectures*. Hilton Head, SC, July. ACM Press, New York.

Blelloch, G. E. and Little, J. J. 1994. Parallel solutions to geometric problems in the scan model of computation. *J. Comput. Syst. Sci.* 48(1):90–115.

Brent, R. P. 1974. The parallel evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.* 21(2):201–206.

Chan, T. M. Y., Snoeyink, J., and Yap, C. K. 1995. Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three, pp. 282–291. In *Proc. 6th Annu. ACM–SIAM Symp. Discrete Algorithms*. ACM–SIAM, ACM Press, New York.

Cole, R. 1988. Parallel merge sort. *SIAM J. Comput.* 17(4):770–785.

Cook, S. A. 1981. Towards a complexity theory of synchronous parallel computation. *Enseignement Mathematique* 27:99–124.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. 1993. LogP: towards a realistic model of parallel computation, pp. 1–12. In *Proc. 4th ACM SIGPLAN Symp. Principles Pract. Parallel Programming*. ACM Press, New York.

Cypher, R. and Sanz, J. L. C. 1994. *The SIMD Model of Parallel Computation*. Springer–Verlag, New York.

Fortune, S. and Wyllie, J. 1978. Parallelism in random access machines, pp. 114–118. In *Proc. 10th Annu. ACM Symp. Theory Comput.* ACM Press, New York.

Gazit, H. 1991. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.* 20(6):1046–1067.

Gibbons, P. B., Matias, Y., and Ramachandran, V. 1994. The QRQW PRAM: accounting for contention in parallel algorithms, pp. 638–648. In *Proc. 5th Annu. ACM–SIAM Symp. Discrete Algorithms*. Jan. ACM Press, New York.

Gibbons, A. and Ritter, W. 1990. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, England.

Goldshlager, L. M. 1978. A unified approach to models of synchronous parallel machines, pp. 89–94. In *Proc. 10th Annu. ACM Symp. Theory Comput.* ACM Press, New York.

Goodrich, M. T. 1996. Parallel algorithms in geometry. In *CRC Handbook of Discrete and Computational Geometry*. CRC Press, Boca Raton, FL.

Greiner, J. 1994. A comparison of data-parallel algorithms for connected components, pp. 16–25. In *Proc. 6th Annu. ACM Symp. Parallel Algorithms Architectures.* June. ACM Press, New York.

Halperin, S. and Zwick, U. 1994. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM, pp. 1–10. In *Proc. ACM Symp. Parallel Algorithms Architectures*. June. ACM Press, New York.

Harris, T. J. 1994. A survey of pram simulation techniques. *ACM Comput. Surv.* 26(2):187–206.

Huang, J. S. and Chow, Y. C. 1983. Parallel sorting and data partitioning by sampling, pp. 627–631. In *Proc. IEEE Comput. Soc. 7th Int. Comput. Software Appl. Conf.* Nov.

JáJá, J. 1992. *An Introduction to Parallel Algorithms*. Addison–Wesley, Reading, MA.

Karlin, A. R. and Upfal, E. 1988. Parallel hashing: an efficient implementation of shared memory. *J. Assoc. Comput. Mach.* 35:876–892.

Karp, R. M. and Ramachandran, V. 1990. Parallel algorithms for shared memory machines. In *Handbook of Theoretical Computer Science — Volume A: Algorithms and Complexity*. J. Van Leeuwen, ed. MIT Press, Cambridge, MA.

Kirkpatrick, D. G. and Seidel, R. 1986. The ultimate planar convex hull algorithm? *SIAM J. Comput.* 15:287–299.

Klein, P. N. and Tarjan, R. E. 1994. A randomized linear-time algorithm for finding minimum spanning trees. In *Proc. ACM Symp. Theory Comput.* May. ACM Press, New York.

Knuth, D. E. 1973. *Sorting and Searching. Vol. 3. The Art of Computer Programming*. Addison–Wesley, Reading, MA.

Kogge, P. M. and Stone, H. S. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.* C-22(8):786–793.

Kumar, V., Grama, A., Gupta, A., and Karypis, G. 1994. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin Cummings, Redwood City, CA.

Leighton, F. T. 1992. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*. Morgan Kaufmann, San Mateo, CA.

Leiserson, C. E. 1985. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* C-34(10):892–901.

Luby, M. 1985. A simple parallel algorithm for the maximal independent set problem, pp. 1–10. In *Proc. ACM Symp. Theory Comput.* May. ACM Press, New York.

Maon, Y., Schieber, B., and Vishkin, U. 1986. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theor. Comput. Sci.* 47:277–298.

Matias, Y. and Vishkin, U. 1991. On parallel hashing and integer sorting. *J. Algorithms* 12(4):573–606.

Miller, G. L. and Ramachandran, V. 1992. A new graph triconnectivity algorithm and its parallelization. *Combinatorica* 12(1):53–76.

Miller, G. and Reif, J. 1989. Parallel tree contraction part 1: fundamentals. In *Randomness and Computation. Vol. 5. Advances in Computing Research*, pp. 47–72. JAI Press, Greenwich, CT.

Miller, G. L. and Reif, J. H. 1991. Parallel tree contraction part 2: further applications. *SIAM J. Comput.* 20(6):1128–1147.

Overmars, M. H. and Van Leeuwen, J. 1981. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.* 23:166–204.

Padua, D., Gelernter, D., and Nicolau, A., eds. 1990. *Languages and Compilers for Parallel Computing Research Monographs in Parallel and Distributed*. MIT Press, Cambridge, MA.

Pippenger, N. 1979. On simultaneous resource bounds, pp. 307–311. In *Proc. 20th Annu. Symp. Found. Comput. Sci.*

Pratt, V. R. and Stockmeyer, L. J. 1976. A characterization of the power of vector machines. *J. Comput. Syst. Sci.* 12:198–221.

Preparata, F. P. and Shamos, M. I. 1985. *Computational Geometry — An Introduction*. Springer–Verlag, New York.

Ranade, A. G. 1991. How to emulate shared memory. *J. Comput. Syst. Sci.* 42(3):307–326.

Reid-Miller, M. 1994. List ranking and list scan on the Cray C-90, pp. 104–113. In *Proc. 6th Annu. ACM Symp. Parallel Algorithms Architectures*. June. ACM Press, New York.

Reif, J. H., ed. 1993. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, CA.

Reif, J. H. and Valiant, L. G. 1983. A logarithmic time sort for linear size networks, pp. 10–16. In *Proc. 15th Annu. ACM Symp. Theory Comput.* April. ACM Press, New York.

Savitch, W. J. and Stimson, M. 1979. Time bounded random access machines with parallel processing. *J. Assoc. Comput. Mach.* 26:103–118.

Shiloach, Y. and Vishkin, U. 1981. Finding the maximum, merging and sorting in a parallel computation model. *J. Algorithms* 2(1):88–102.

Shiloach, Y. and Vishkin, U. 1982. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms* 3:57–67.

Stone, H. S. 1975. Parallel tridiagonal equation solvers. *ACM Trans. Math. Software* 1(4):289–307.

Strassen, V. 1969. Gaussian elimination is not optimal. *Numerische Mathematik* 14(3):354–356.

Tarjan, R. E. and Vishkin, U. 1985. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.* 14(4):862–874.

Valiant, L. G. 1990a. A bridging model for parallel computation. *Commun. ACM* 33(8):103–111.

Valiant, L. G. 1990b. General purpose parallel architectures, pp. 943–971. In *Handbook of Theoretical Computer Science*. J. van Leeuwen, ed. Elsevier Science, B. V., Amsterdam, The Netherlands.

Vishkin, U. 1984. Parallel-design distributed-implementation (PDDI) general purpose computer. *Theor. Comp. Sci.* 32:157–172.

Wyllie, J. C. 1979. The Complexity of Parallel Computations. *Department of Computer Science, Tech. Rep.* TR-79-387, Cornell University, Ithaca, NY. Aug.

11 Computational Geometry

D. T. Lee
Northwestern University

11.1 Introduction

Computational geometry evolves from the classical discipline of design and analysis of algorithms, and has received a great deal of attention in the past two decades since its identification in 1975 by Shamos. It is concerned with the computational complexity of geometric problems that arise in various disciplines such as pattern recognition, computer graphics, computer vision, robotics, very large-scale integrated (VLSI) layout, operations research, statistics, etc. In contrast with the classical approach to proving mathematical theorems about geometry-related problems, this discipline emphasizes the computational aspect of these problems and attempts to exploit the underlying geometric properties possible, e.g., the metric space, to derive efficient algorithmic solutions.

The classical theorem, for instance, that a set S is convex if and only if for any $0 \leq \alpha \leq 1$ the convex combination $\alpha p + (1 - \alpha)q = r$ is in S for any pair of elements $p, q \in S$, is very fundamental in establishing convexity of a set. In geometric terms, a body S in the Euclidean space is convex if and only if the line segment joining any two points in S lies totally in S. But this theorem per se is not suitable for computational purposes as there are infinitely many possible pairs of points to be considered. However, other properties of convexity can be utilized to yield an algorithm. Consider the following problem. Given a simple closed Jordan polygonal curve, determine if the interior region enclosed by the curve is convex. This problem can be readily solved by observing that if the line segments defined by all pairs of vertices of the polygonal curve, $\overline{v_i, v_j}, i \neq j, 1 \leq i, j \leq n$, where n denotes the total number of vertices, lie totally inside the region, then the region is convex. This would yield a straightforward algorithm with time complexity $O(n^3)$, as there are $O(n^2)$ line segments, and to test if each line segment lies totally in the region takes $O(n)$ time by comparing it against every polygonal segment. As we shall show, this problem can be solved in $O(n)$ time by utilizing other geometric properties.

At this point, an astute reader might have come up with an $O(n)$ algorithm by making the observation: Because the interior angle of each vertex must be strictly less than π in order for the region to be convex,

we just have to check for every consecutive three vertices v_{i-1}, v_i, v_{i+1} that the angle at vertex v_i is less than π. (A vertex whose internal angle has a measure less than π is said to be *convex*; otherwise, it is said to be *reflex*.) One may just be content with this solution. Mathematically speaking, this solution is fine and indeed runs in $O(n)$ time. The problem is that the algorithm implemented in this straightforward manner without care may produce an incorrect answer when the input polygonal curve is ill formed. That is, if the input polygonal curve is not simple, i.e., it self-intersects, then the *enclosed* region by this closed curve is not well defined. The algorithm, without checking this simplicity condition, may produce a wrong answer. Note that the preceding observation that all of the vertices must be convex in order to have a convex region is only a necessary condition. Only when the input polygonal curve is *verified* to be simple will the algorithm produce a correct answer. But to verify whether the input polygonal curve self-intersects or not is no longer as straightforward. The fact that we are dealing with computer solutions to geometric problems may make the task of designing an algorithm and proving its correctness nontrivial.

An objective of this discipline in the theoretical context is to prove lower bounds of the complexity of geometric problems and to devise algorithms (giving upper bounds) whose complexity *matches* the lower bounds. That is, we are interested in the *intrinsic* difficulty of geometric computational problems under a certain computation model and at the same time are concerned with the algorithmic solutions that are provably optimal in the worst or average case. In this regard, the asymptotic time (or space) complexity of an algorithm is of interest. Because of its applications to various science and engineering related disciplines, researchers in this field have begun to address the efficacy of the algorithms, the issues concerning robustness and numerical stability [Fortune 1993], and the actual running times of their implementions.

In this chapter, we concentrate mostly on the theoretical development of this field in the context of sequential computation. Parallel computation geometry is beyond the scope of this chapter. We will adopt the *real* random access machine (RAM) model of computation in which all arithmetic operations, comparisons, kth-root, exponential or logarithmic functions take unit time. For more details refer to Edelsbrunner [1987], Mulmuley [1994], and Preparata and Shamos [1985]. We begin with a summary of problem solving techniques that have been developed [Lee and Preparata 1982, O'Rourke 1994, Yao 1994] and then discuss a number of topics that are central to this field, along with additional references for further reading about these topics.

11.2 Problem Solving Techniques

We give an example for each of the eight major problem-solving paradigms that are prevalent in this field. In subsequent sections we make reference to these techniques whenever appropriate.

11.2.1 Incremental Construction

This is the simplest and most intuitive method, also known as *iterative method*. That is, we compute the solution in an iterative manner by considering the input incrementally.

Consider the problem of computing the line arrangements in the plane. Given is a set $\mathcal{L}$ of n straight lines in the plane, and we want to compute the partition of the plane induced by $\mathcal{L}$. One obvious approach is to compute the partition iteratively by considering one line at a time [Chazelle et al. 1985]. As shown in Figure 11.1, when line i is inserted, we need to traverse the regions that are intersected by the line and construct the new partition at the same time. One can show that the traversal and repartitioning of the intersected regions can be done in $O(n)$ time per insertion, resulting in a total of $O(n^2)$ time. This algorithm is asymptotically optimal because the running time is proportional to the amount of space required to represent the partition. This incremental approach also generalizes to higher dimensions. We conclude with the theorem [Edelsbrunner et al. 1986].

Theorem 11.1 *The problem of computing the arrangement $\mathcal{A}(H)$ of a set H of n hyperplanes in $\Re^k$ can be solved iteratively in $O(n^k)$ time and space, which is optimal.*

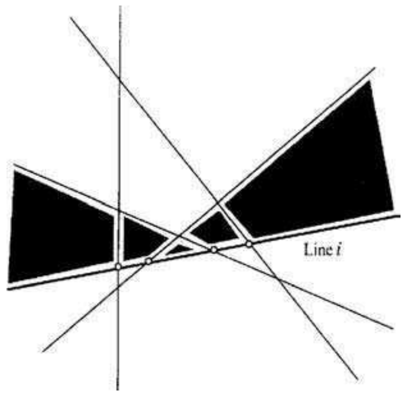


FIGURE 11.1 Incremental construction of line arrangement: phase i.

11.2.2 Plane Sweep

This approach works most effectively for two-dimensional problems for which the solution can be computed incrementally as the entire input is scanned in a certain order. The concept can be easily generalized to higher dimensions [Bieri and Nef 1982]. This is also known as the *scan-line* method in computer graphics and is used for a variety of applications such as shading and polygon filling, among others.

Consider the problem of computing the *measure* of the union of n isothetic rectangles, i.e., whose sides are parallel to the coordinate axes. We would proceed with a *vertical* sweep line, sweeping across the plane from left to right. As we sweep the plane, we need to keep track of the rectangles that intersect the current sweep line and those that are yet to be visited. In the meantime we compute the area covered by the union of the rectangles seen so far. More formally, associated with this approach there are two basic data structures containing all *relevant* information that should be maintained.

1. *Event schedule* defines a sequence of *event points* that the sweep-line status will change. In this example, the sweep-line status will change only at the left and right boundary edges of each rectangle.
2. *Sweep-line status* records the information of the geometric structure that is being swept. In this example the sweep-line status keeps track of the set of rectangles intersecting the current sweep line.

The event schedule is normally represented by a *priority queue*, and the list of events may change dynamically. In this case, the events are static; they are the x-coordinates of the left and right boundary edges of each rectangle. The sweep-line status is represented by a suitable data structure that supports insertions, deletions, and computation of the partial solution at each event point. In this example a *segment tree* attributed to Bentley is sufficient [Preparata and Shamos 1985]. Because we are computing the area of the rectangles, we need to be able to know the *new* area covered by the current sweep line between two adjacent event points. Suppose at event point x_{i-1} we maintain a partial solution $\mathcal{A}_{i-1}$. In Figure 11.2 the shaded area S needs to be added to the partial solution, that is, $\mathcal{A}_i = \mathcal{A}_{i-1} + S$. The shaded area is equal to the total measure, denoted sum_ℓ, of the union of vertical line segments representing the intersection of the rectangles and the current sweep line times the distance between the two event points x_i and x_{i-1}. If the next event corresponds to the left boundary of a rectangle, the corresponding vertical segment, $\overline{p,q}$ in Figure 11.2, needs to be inserted to the segment tree. If the next event corresponds to a right boundary edge, the segment, $\overline{u,v}$ needs to be deleted from the segment tree. In either case, the total measure sum_ℓ should be updated accordingly. The correctness of this algorithm can be established by observing that the partial solution obtained for the rectangles to the *left* of the sweep line is maintained correctly. In fact, this property is typical of any algorithm based on the plane-sweep technique.

Because the segment tree structure supports segment insertions and deletions and the update (of sum_ℓ) operation in $O(\log n)$ time per event point, the total amount of time needed is $O(n \log n)$.

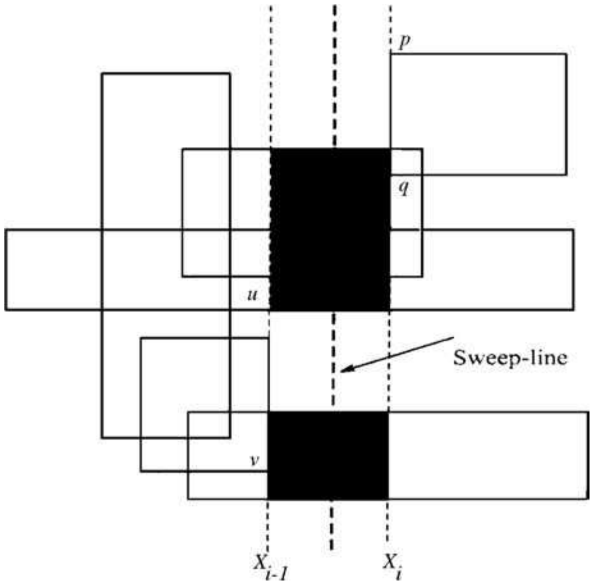


FIGURE 11.2 The plane-sweep approach to the measure problem in two dimensions.

The measure of the union of rectangles in higher dimensions also can be solved by the plane-sweep technique with quad trees, a generalization of segment trees.

Theorem 11.2 *The problem of computing the measure of n isothetic rectangles in k dimensions can be solved in $O(n \log n)$ time, for $k \leq 2$ and in $O(n^{k-1})$ time for $k \geq 3$.*

The time bound is asymptotically optimal. Even in one dimension, i.e., computing the total length of the union of n intervals requires $\Omega(n \log n)$ time (see Preparata and Shamos [1985]).

We remark that the sweep line used in this approach is not necessarily a straight line. It can be a topological line as long as the objects stored in the sweep line status are ordered, and the method is called *topological sweep* [Asano et al. 1994, Edelsbrunner and Guibas 1989]. Note that the measure of isothetic rectangles can also be solved using the divide-and-conquer paradigm to be discussed.

11.2.3 Geometric Duality

This is a geometric transformation that maps a given problem into its equivalent form, preserving certain geometric properties so as to manipulate the objects in a more convenient manner. We will see its usefulness for a number of problems to be discussed. Here let us describe a transformation in k-dimensions, known as *polarity* or *duality*, denoted $\mathcal{D}$, that maps d-dimensional varieties to $(k-1-d)$-dimensional varieties, $0 \leq d < k$.

Consider any point $p = (\pi_1, \pi_2, \ldots, \pi_k) \in \Re^k$ other than the origin. The dual of p, denoted $\mathcal{D}(p)$, is the hyperplane $\pi_1 x_1 + \pi_2 x_2 + \cdots + \pi_k x_k = 1$. Similarly, a hyperplane that does not contain the origin is mapped to a point such that $\mathcal{D}(\mathcal{D}(p)) = p$. Geometrically speaking, point p is mapped to a hyperplane whose normal is the vector determined by p and the origin and whose distance to the origin is the reciprocal of that between p and the origin. Let S denote the unit sphere $S : x_1^2 + x_2^2 + \cdots + x_k^2 = 1$. If point p is external to S, then it is mapped to a hyperplane $\mathcal{D}(p)$ that intersects S at those points q that admit supporting hyperplanes h such that $h \cap S = q$ and $p \in h$. In two dimensions a point p outside of the unit disk will be mapped to a line intersecting the disk at two points, q_1 and q_2, such that line segments $\overline{p, q_1}$ and $\overline{p, q_2}$ are tangent to the disk. Note that the distances from p to the origin and from the line $\mathcal{D}(p)$ to the origin are reciprocal to each other. Figure 11.3a shows the duality transformation in two dimensions. In

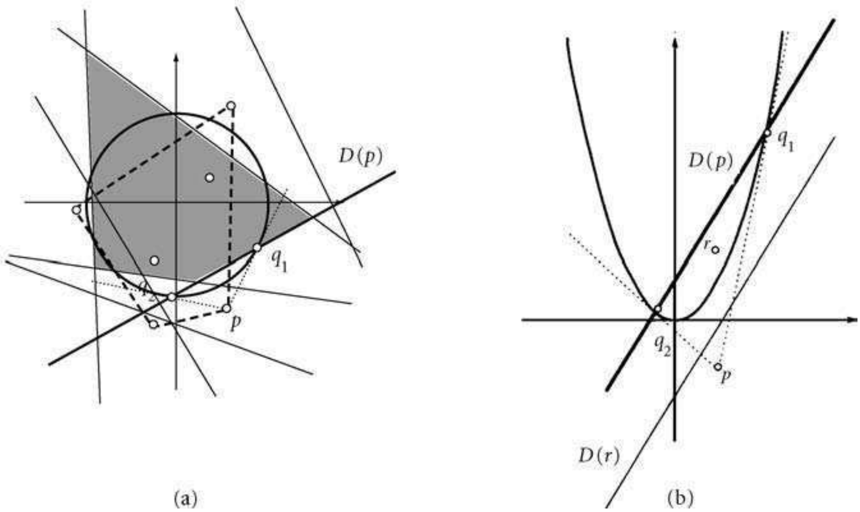


FIGURE 11.3 Geometric duality transformation in two dimensions.

particular, point p is mapped to the line shown in boldface. For each hyperplane $\mathcal{D}(p)$, let $\mathcal{D}(p)^+$ denote the half-space that contains the origin and let $\mathcal{D}(p)^-$ denote the other half-space.

The duality transformation not only leads to dual arrangements of hyperplanes and configurations of points and vice versa, but also preserves the following properties.

Incidence: Point p belongs to hyperplane h if and only if point $\mathcal{D}(\langle)$ belongs to hyperplane $\mathcal{D}(p)$.

Order: Point p lies in half-space h^+ (respectively, h^-) if and only if point $\mathcal{D}(\langle)$ lies in half-space $\mathcal{D}(p)^+$ (respectively, $\mathcal{D}(p)^-$).

Figure 11.3a shows the convex hull of a set of points that are mapped by the duality transformation to the shaded region, which is the common intersection of the half-planes $\mathcal{D}(p)^+$ for all points p.

Another transformation using the unit paraboloid U, represented as $U: x_k = x_1^2 + x_2^2 + \cdots + x_{k-1}^2$, can also be similarly defined. That is, point $p = (\pi_1, \pi_2, \ldots, \pi_k) \in R^k$ is mapped to a hyperplane $\mathcal{D}_{\sqcap}(\surd)$ represented by the equation $x_k = 2\pi_1 x_1 + 2\pi_2 x_2 + \cdots + 2\pi_{k-1}x_{k-1} - \pi_k$. And each nonvertical hyperplane is mapped to a point in a similar manner such that $\mathcal{D}_u(\mathcal{D}_u(p)) = p$. Figure 11.3b illustrates the two-dimensional case, in which point p is mapped to a line shown in boldface. For more details see, e.g., Edelsbrunner [1987] and Preparata and Shamos [1985].

11.2.4 Locus

This approach is often used as a preprocessing step for a geometric *searching* problem to achieve faster query-answering response time. For instance, given a *fixed* database consisting of geographical locations of post offices, each represented by a point in the plane, one would like to be able to efficiently answer queries of the form: "what is the nearest post office to location q?" for some query point q. The locus approach to this problem is to partition the plane into n regions, each of which consists of the locus of *query* points for which the *answer* is the same. The partition of the plane is the so-called *Voronoi* diagram discussed subsequently. In Figure 11.7, the post office closest to query point q is site s_i. Once the Voronoi diagram is available, the query problem reduces to that of locating the region that contains the query, an instance of the point-location problem discussed in Section 11.3.

11.2.5 Divide-and-Conquer

This is a classic problem-solving technique and has proven to be very powerful for geometric problems as well. This technique normally involves partitioning of the given problem into several subproblems,

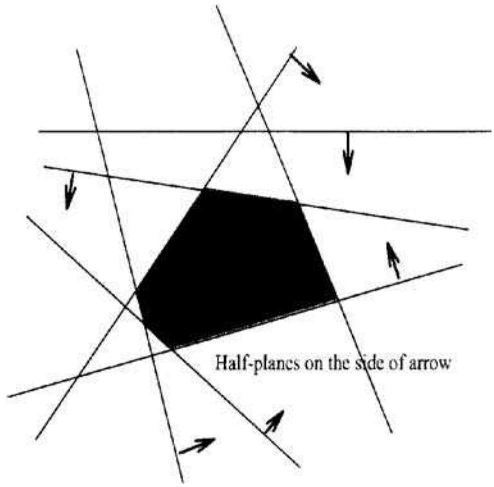


FIGURE 11.4 The common intersection of half-planes.

recursively solving each subproblem, and then combining the solutions to each of the subproblems to obtain the final solution to the original problem. We illustrate this paradigm by considering the problem of computing the common intersection of n half-planes in the plane. Given is a set S of n half-planes, h_i, represented by $a_i x + b_i y \leq c_i, i = 1, 2, \ldots, n$. It is well known that the common intersection of half-planes, denoted $CI(S) = \bigcap_{i=1}^{n} h_i$, is a convex set, which may or may not be bounded. If it is bounded, it is a convex polygon. See Figure 11.4, in which the shaded area is the common intersection.

The divide-and-conquer paradigm consists of the following steps.

Algorithm Common_Intersection_D&C (S)

1. *If* $|S| \leq 3$, *compute the intersection* $CI(S)$ *explicitly*. **Return** $(CI(S))$.
2. *Divide S into two approximately equal subsets* S_1 *and* S_2.
3. $CI(S_1) =$ **Common_Intersection_D&C**(S_1).
4. $CI(S_2) =$ **Common_Intersection_D&C**(S_2).
5. $CI(S) =$ **Merge**$(CI(S_1), CI(S_2))$.
6. **Return** $(CI(S))$.

The key step is the *merge* of two common intersections. Because $CI(S_1)$ and $CI(S_2)$ are convex, the merge step basically calls for the computation of the intersection of two convex polygons, which can be solved in time proportional to the size of the polygons (cf. subsequent section on intersection). The running time of the divide-and-conquer algorithm is easily shown to be $O(n \log n)$, as given by the following recurrence formula, where $n = |S|$:

$$T(3) = O(1)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) + M\left(\frac{n}{2}, \frac{n}{2}\right)$$

where $M(n/2, n/2) = O(n)$ denotes the merge time (step 5).

Theorem 11.3 *The common intersection of n half-planes can be solved in* $O(n \log n)$ *time by the divide-and-conquer method.*

The time complexity of the algorithm is asymptotically optimal, as the problem of sorting can be reduced to it [Preparata and Shamos 1985].

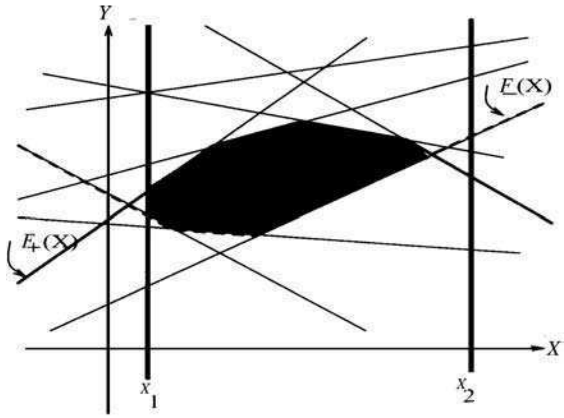


FIGURE 11.5 Feasible region defined by upward- and downward-convex piecewise linear functions.

11.2.6 Prune-and-Search

This approach, developed by Dyer [1986] and Megiddo [1983a, 1983b, 1984], is a very powerful method for solving a number of geometric optimization problems, one of which is the well-known linear programming problem. Using this approach, they obtained an algorithm whose running time is linear in the number of constraints. For more development of linear programming problems, see Megiddo [1983c, 1986]. The main idea is to prune away a fraction of *redundant* input constraints in each iteration while searching for the solution. We use a two-dimensional linear programming problem to illustrate this approach. Without loss of generality, we consider the following linear programming problem:

$$\begin{aligned} &\text{Minimize} && Y \\ &\text{subject to} && \alpha_i X + \beta_i Y + \gamma_i \le 0, \quad i = 1, 2, \ldots, n \end{aligned}$$

These n constraints are partitioned into three classes, C_0, C_+, C_-, depending on whether β_i is zero, positive, or negative, respectively. The constraints in class C_0 define an X-interval $[x_1, x_2]$, which constrains the solution, if any. The constraints in classes C_+ and C_- define, however, upward- and downward-convex piecewise linear functions $F_+(X)$ and $F_-(X)$ delimiting the feasible region* (Figure 11.5). The problem now becomes

$$\begin{aligned} &\text{Minimize} && F_-(X) \\ &\text{subject to} && F_-(X) \le F_+(X) \\ &&& x_1 \le X \le x_2 \end{aligned}$$

Let λ^* denote the optimal solution, if it exists. The values of $F_-(\lambda)$ and $F_+(\lambda)$ for any λ can be computed in $O(n)$ time, based on the slopes $-\alpha_i/\beta_i$. Thus, in $O(n)$ time one can determine for any $\lambda' \in [x_1, x_2]$ if (1) λ' is infeasible, and there is no solution, (2) λ' is infeasible, and we know a feasible solution is less or greater than λ', (3) $\lambda' = \lambda^*$, or (4) λ' is feasible, and whether λ^* is less or greater than λ'.

To choose λ' we partition constraints in classes C_- and C_+ into pairs and find the abscissa $\lambda_{i,j}$ of their intersection. If $\lambda_{i,j} \notin [x_1, x_2]$ then one of the constraints can be eliminated as redundant. For those $\lambda_{i,j}$ that are in $[x_1, x_2]$ we find in $O(n)$ time [Dobkin and Munro 1981] the median $\lambda'_{i,j}$ and compute $F_-(\lambda'_{i,j})$ and $F_+(\lambda'_{i,j})$. By the preceding arguments that we can determine where λ^* should lie, we know one-half of the $\lambda_{i,j}$ do not lie in the region containing λ^*. Therefore, one constraint of the corresponding pair can

*These upward- and downward-convex functions are also known as the upper and lower *envelopes* of the line arrangements for lines belonging to classes C_- and C_+, respectively.

be eliminated. The process iterates. In other words, in each iteration at least a fixed fraction $\delta = 1/4$ of the current constraints can be eliminated. Because each iteration takes $O(n)$ time, the total time spent is $Cn + C\delta n + \cdots = O(n)$. In higher dimensions, we have the following result due to Dyer [1986] and Clarkson [1986].

Theorem 11.4 *A linear program in k-dimensions with n constraints can be solved in $O(3^{k^2}n)$ time.*

We note here some of the new recent developments for linear programming. There are several randomized algorithms for this problem, of which the best expected complexity, $O(k^2n + k^{k/2+O(1)}\log n)$ is due to Clarkson [1988], which is later improved by Matoušek et al. [1992] to run in $O(k^2n + e^{O(\sqrt{k \ln k})}\log n)$. Clarkson's [1988] algorithm is applicable to work in a general framework, which includes various other geometric optimization problems, such as *smallest enclosing ellipsoid*. The best known deterministic algorithm for linear programming is due to Chazelle and Matoušek [1993], which runs in $O(k^{7k+o(k)}n)$ time.

11.2.7 Dynamization

Techniques have been developed for query-answering problems, classified as *geometric searching* problems, in which the underlying database is changing over (discrete) time. A typical geometric searching problem is the *membership* problem, i.e., given a set $\mathcal{D}$ of objects, determine if x is a member of $\mathcal{D}$, or the *nearest neighbor searching* problem, i.e., given a set $\mathcal{D}$ of objects, find an object that is closest to x according to some distance measure. In the database area, these two problems are referred to as the *exact match* and *best match* queries. The idea is to make use of good data structures for a static database and enhance them with dynamization mechanisms so that updates of the database can be accommodated on line and yet queries to the database can be answered efficiently.

A general query Q contains a variable of type $T1$ and is asked of a set of objects of type $T2$. The answer to the query is of type $T3$. More formally, Q can be considered as a mapping from $T1$ and subsets of $T2$ to $T3$, that is, $Q : T1 \times 2^{T2} \rightarrow T3$. The class of geometric searching problems to which the dynamization techniques are applicable is the class of *decomposable searching problems* [Bentley and Saxe 1980].

Definition 11.1 A searching problem with query Q is decomposable if there exists an efficiently computable associative, and communtative binary operator @ satisfying the condition

$$Q(x, A \cup B) = @(Q(x, A), Q(x, B))$$

In other words, the answer to a query Q in $\mathcal{D}$ can be computed by the answers to two subsets $\mathcal{D}_\infty$ and $\mathcal{D}_\in$ of $\mathcal{D}$. The membership problem and the nearest-neighbor searching problem previously mentioned are decomposable.

To answer queries efficiently, we have a data structure to support various update operations. There are typically three measures to evaluate a static data structure $\mathcal{A}$. They are:

1. $P_\mathcal{A}(N)$, the preprocessing time required to build $\mathcal{A}$
2. $S_\mathcal{A}(N)$, the storage required to represent $\mathcal{A}$
3. $Q_\mathcal{A}(N)$, the query response time required to search in $\mathcal{A}$

where N denotes the number of elements represented in $\mathcal{A}$. One would add another measure $U_\mathcal{A}(N)$ to represent the *update* time.

Consider the nearest-neighbor searching problem in the Euclidean plane. Given a set of n points in the plane, we want to find the nearest neighbor of a query point x. One can use the Voronoi diagram data structure $\mathcal{A}$ (cf. subsequent section on Voronoi diagrams) and point location scheme (cf. subsequent section on point location) to achieve the following: $P_\mathcal{A}(n) = O(n \log n)$, $S_\mathcal{A}(n) = O(n)$, and $Q_\mathcal{A}(n) = O(\log n)$. We now convert the static data structure $\mathcal{A}$ to a dynamic one, denoted $\mathcal{D}$, to support insertions and deletions

as well. There are a number of dynamization techniques, but we describe the technique developed by van Leeuwan and Wood [1980] that provides the general flavor of the approach.

The general principle is to decompose $\mathcal{A}$ into a collection of separate data structures so that each update can be confined to one or a small, fixed number of them; however, to avoid degrading the query response time we cannot afford to have excessive fragmentation because queries involve the entire collection.

Let $\{x_k\}_{k\geq 1}$ be a sequence of increasing integers, called *switch points*, where x_k is divisible by k and $x_{k+1}/(k+1) > x_k/k$. Let $x_0 = 0, y_k = x_k/k$, and n denote the current size of the point set. For a given *level* k, $\mathcal{D}$ consists of $(k+1)$ static structures of the same type, one of which, called *dump* is designated to allow for insertions. Each substructure $\mathcal{B}$ has size $y_k \leq s(\mathcal{B}) \leq \dagger_{\|+\infty}$, and the dump has size $0 \leq s(dump) < y_{k+1}$. A block $\mathcal{B}$ is called *low* or *full* depending on whether $s(\mathcal{B}) = \dagger_{\|}$ or $s(\mathcal{B}) = \dagger_{\|+\infty}$, respectively, and is called *partial* otherwise. When an insertion to the dump makes its size equal to y_{k+1}, it becomes a full block and any nonfull block can be used as the dump. If all blocks are full, we switch to the next level. Note that at this point the total size is $y_{k+1} * (k+1) = x_{k+1}$. That is, at the beginning of level $k+1$, we have $k+1$ low blocks and we create a new dump, which has size 0. When a deletion from a low block occurs, we need to borrow an element either from the dump, if it is not empty, or from a partial block. When all blocks are low and $s(dump) = 0$, we switch to level $k-1$, making the low block from which the latest deletion occurs the *dump*. The level switching can be performed in $O(1)$ time. We have the following:

Theorem 11.5 *Any static data structure $\mathcal{A}$ used for a decomposable searching problem can be transformed into a dynamic data structure $\mathcal{D}$ for the same problem with the following performance. For $x_k \leq n < x_{k+1}$, $Q_{\mathcal{D}}(n) = O(kQ_{\mathcal{A}}(y_{k+1}))$, $U_{\mathcal{D}}(n) = O(C(n)+U_{\mathcal{A}}(y_{k+1}))$, and $S_{\mathcal{D}}(n) = O(kS_{\mathcal{A}}(y_{k+1}))$, where $C(n)$ denotes the time needed to look up the block which contains the data when a deletion occurs.*

If we choose, for example, x_k to be the first multiple of k greater than or equal to 2^k, that is, $k = \log_2 n$, then y_k is about $n/\log_2 n$. Because we know there exists an $\mathcal{A}$ with $Q_{\mathcal{A}}(n) = O(\log n)$ and $U_{\mathcal{A}}(n) = P_{\mathcal{A}}(n) = O(n \log n)$, we have the following corollary.

Corollary 11.1 The nearest-neighbor searching problem in the plane can be solved in $O(\log^2 n)$ query time and $O(n)$ update time. [Note that $C(n)$ in this case is $O(\log n)$.]

There are other dynamization schemes that exhibit various query-time/space and query-time/update-time tradeoffs. The interested reader is referred to Chiang and Tamassia [1992], Edelsbrunner [1987], Mehlhorn [1984], Overmars [1983], and Preparata and Shamos [1985] for more information.

11.2.8 Random Sampling

Randomized algorithms have received a great deal of attention recently because of their potential applications. See Chapter 4 for more information. For a variety of geometric problems, randomization techniques help in building geometric subdivisions and data structures to quickly answer queries about such subdivisions. The resulting randomized algorithms are simpler to implement and/or asymptotically faster than those previously known. It is important to note that the focus of randomization is *not* on random input, such as a collection of points randomly chosen uniformly and independently from a region. We are concerned with algorithms that use a source of random numbers and analyze their performance for an arbitrary input. Unlike *Monte Carlo* algorithms, whose output may be incorrect (with very low probability), the randomized algorithms, known as *Las Vegas* algorithms, considered here are guaranteed to produce a correct output.

There are a good deal of newly developed randomized algorithms for geometric problems. See Du and Hwang [1992] for more details. Randomization gives a general way to divide and conquer geometric problems and can be used for both parallel and serial computation. We will use a familiar example to illustrate this approach.

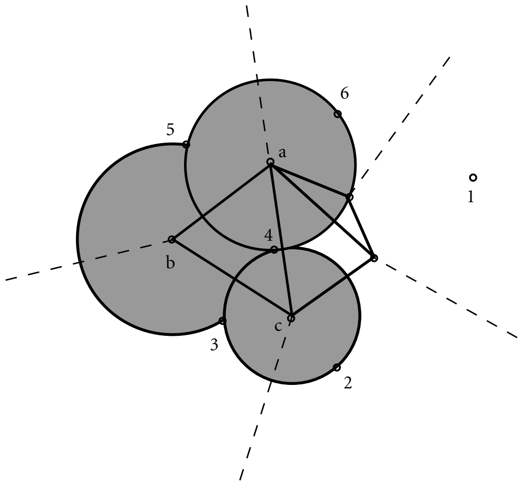


FIGURE 11.6 A triangulation of the Voronoi diagram of six sites and $K_{\mathcal{R}}(T)$, $T = \Delta(a, b, c)$.

Let us consider the problem of nearest-neighbor searching discussed in the preceding subsection. Let $\mathcal{D}$ be a set of n points in the plane and q be the query point. A simple approach to this problem is:

Algorithm ***S***

- *Compute the distance to* q *for each point* $p \in \mathcal{D}$.
- *Return the point* p *whose distance is the smallest.*

It is clear that Algorithm S, requiring $O(n)$ time, is not suitable if we need to answer many queries of this type. To obtain faster query response time one can use the technique discussed in the preceding subsection. An alternative is to use the *random sampling* technique as follows. We pick a random sample, a subset $\mathcal{R} \subset \mathcal{D}$ of size r. Let point $p \in \mathcal{R}$ be the nearest neighbor of q in $\mathcal{R}$. The open disk $K_{\mathcal{R}}(q)$ centered at q and passing through p does not contain any other point in $\mathcal{R}$. The answer to the query is either p or some point of $\mathcal{D}$ that lies in $K_{\mathcal{R}}(q)$.

We now extend the above observation to a finite region G in the plane. Let $K_{\mathcal{R}}(G)$ be the union of disks $K_{\mathcal{R}}(r)$ for all $r \in G$. If a query q lies in G, the nearest neighbor of q must be in $K_{\mathcal{R}}(G)$ or in $\mathcal{R}$. Let us consider the Voronoi diagram, $\mathcal{V}(\mathcal{R})$ of $\mathcal{R}$ and a triangulation, $\Delta(\mathcal{V}(\mathcal{R}))$. For each triangle T with vertices a, b, c of $\Delta(\mathcal{V}(\mathcal{R}))$ we have $K_{\mathcal{R}}(T) = K_{\mathcal{R}}(a) \cup K_{\mathcal{R}}(b) \cup K_{\mathcal{R}}(c)$, shown as the shaded area in Figure 11.6. A probability lemma [Clarkson 1988] shows that with probability at least $1 - O(1/n^2)$ the candidate set $\mathcal{D} \cap K_{\mathcal{R}}(T)$ for all $T \in \Delta(\mathcal{V}(\mathcal{R}))$ contains $O(\log n)n/r$ points. More precisely, if $r > 5$ then with probability at least $1 - e^{-C/2+3\ell n r}$ each open disk $K_{\mathcal{R}}(r)$ for $r \in \mathcal{R}$ contains no more than Cn/r points of $\mathcal{D}$. If we choose r to be $\sqrt{n}$, the query time becomes $O(\sqrt{n}\log n)$, a speedup from Algorithm S. If we apply this scheme recursively to the candidate sets of $\Delta(\mathcal{V}(\mathcal{R}))$, we can get a query time $O(\log n)$ [Clarkson 1988].

There are many applications of these random sampling techniques. Derandomized algorithms were also developed. See, e.g., Chazelle and Friedman [1990] for a deterministic view of random sampling and its use in geometry.

11.3 Classes of Problems

In this section we aim to touch upon classes of problems that are fundamental in this field and describe solutions to them, some of which may be nontrivial. The reader who needs further information about these problems is strongly encouraged to refer to the original articles cited in the references.

11.3.1 Convex Hull

The convex hull of a set of points in $\Re^k$ is the most fundamental problem in computational geometry. Given is a set of points, and we are interested in computing its convex hull, which is defined to be the smallest convex body containing these points. Of course, the first question one has to answer is how to represent the convex hull. An implicit representation is just to list all of the extreme points,* whereas an explicit representation is to list all of the extreme d-faces of dimensions $d = 0, 1, \ldots, k-1$. Thus, the complexity of any convex hull algorithm would have two parts, computation part and the output part. An algorithm is said to be *output sensitive* if its complexity depends on the size of output.

Definition 11.2 The convex hull of a set S of points in $\Re^k$ is the smallest convex set containing S. In two dimensions, the convex hull is a convex polygon containing S; in three dimensions it is a convex polyhedron.

11.3.1.1 Convex Hulls in Two and Three Dimensions

For an arbitrary set of n points in two and three dimensions, we can compute the convex hull using the *Graham scan*, *gift-wrapping*, or *divide-and-conquer* paradigm, which are briefly described next.

Recall that the convex hull of an arbitrary set of points in two dimensions is a convex polygon. The Graham scan computes the convex hull by (1) sorting the input set of points with respect to an interior point, say, O, which is the centroid of the first three noncollinear points, (2) connecting these points into a star-shaped polygon P centered at O, and (3) performing a linear scan to compute the convex hull of the polygon [Preparata and Shamos 1985]. Because step 1 is the dominating step, the Graham scan takes $O(n \log n)$ time.

One can also use the gift-wrapping technique to compute the convex polygon. Starting with a vertex that is known to be on the convex hull, say, the point O, with the smallest y-coordinate, we sweep a half-line emanating from O counterclockwise. The first point v_1 we hit will be the next point on the convex polygon. We then march to v_1, repeat the same process, and find the next vertex v_2. This process terminates when we reach O again. This is similar to wrapping an object with a *rope*. Finding the next vertex takes time proportional to the number of points remaining. Thus, the total time spent is $O(n\mathcal{H})$, where $\mathcal{H}$ denotes the number of points on the convex polygon. The gift-wrapping algorithm is output sensitive and is more efficient than Graham scan if the number of points on the convex polygon is small, that is, $o(\log n)$.

One can also use the divide-and-conquer paradigm. As mentioned previously, the key step is the merge of two convex hulls, each of which is the solution to a subproblem derived from the recursive step. In the division step, we can recursively separate the set into two subsets by a vertical line L. Then the merge step basically calls for computation of two common tangents of these two convex polygons. The computation of the common tangents, also known as *bridges* over line L, begins with a segment connecting the rightmost point l of the left convex polygon to the leftmost point r of the right convex polygon. Advancing the endpoints of this segment in a *zigzag* manner we can reach the top (or the bottom) common tangent such that the entire set of points lies on one side of the line containing the tangent. The running time of the divide-and-conquer algorithm is easily shown to be $O(n \log n)$.

A more sophisticated output-sensitive and optimal algorithm, which runs in $O(n \log \mathcal{H})$ time, has been developed by Kirkpatrick and Seidel [1986]. It is based on a variation of the divide-and-conquer paradigm. The main idea in achieving the optimal result is that of eliminating *redundant* computations. Observe that in the divide-and-conquer approach after the common tangents are obtained, some vertices that used to belong to the left and right convex polygons must be deleted. Had we known these vertices were not on the final convex hull, we could have saved time by not computing them. Kirkpatrick and Seidel capitalized on this concept and introduced the *marriage-before-conquest* principle. They construct the convex hull by

*A point in S is an extreme point if it cannot be expressed as a convex combination of other points in S. In other words, the convex hull of S would change when an extreme point is removed from S.

computing the upper and lower hulls of the set; the computations of these two hulls are symmetric. It performs the *divide* step as usual that decomposes the problem into two subproblems of approximately equal size. Instead of computing the upper hulls recursively for each subproblem, it finds the common tangent segment of the two yet-to-be-computed upper hulls and proceeds recursively. One thing that is worth noting is that the points known not to be on the (convex) upper hull are discarded before the algorithm is invoked recursively. This is the key to obtaining a time bound that is both output sensitive and asymptotically optimal.

The divide-and-conquer scheme can be easily generalized to three dimensions. The merge step in this case calls for computing common supporting faces that *wrap* two recursively computed convex polyhedra. It is observed by Preparata and Hong that the common supporting faces are computed from connecting two *cyclic* sequences of edges, one on each polyhedron [Preparata and Shamos 1985]. The computation of these supporting faces can be accomplished in linear time, giving rise to an $O(n \log n)$ time algorithm. By applying the marriage-before-conquest principle Edelsbrunner and Shi [1991] obtained an $O(n \log^2 \mathcal{H})$ algorithm.

The gift-wrapping approach for computing the convex hull in three dimensions would mimic the process of wrapping a gift with a piece of paper and has a running time of $O(n\mathcal{H})$.

11.3.1.2 Convex Hulls in k-Dimensions, $k > 3$

For convex hulls of higher dimensions, a recent result by Chazelle [1993] showed that the convex hull can be computed in time $O(n \log n + n^{\lfloor k/2 \rfloor})$, which is optimal in all dimensions $k \geq 2$ in the worst case. But this result is insensitive to the output size. The gift-wrapping approach generalizes to higher dimensions and yields an output-sensitive solution with running time $O(n\mathcal{H})$, where $\mathcal{H}$ is the total number of i-faces, $i = 0, 1, \ldots, k-1$, and $\mathcal{H} = O(n^{\lfloor k/2 \rfloor})$ [Edelsbrunner 1987]. One can also use the *beneath-beyond* method of adding points one at a time in ascending order along one of the coordinate axes.* We compute the convex hull $CH(S_{i-1})$ for points $S_{i-1} = \{p_1, p_2, \ldots, p_{i-1}\}$. For each added point p_i, we update $CH(S_{i-1})$ to get $CH(S_i)$, for $i = 2, 3, \ldots, n$, by deleting those t-faces, $t = 0, 1, \ldots, k-1$, that are internal to $CH(S_{i-1} \cup \{p_i\})$. It is shown by Seidel (see Edelsbrunner [1987])that $O(n^2 + \mathcal{H} \log n)$ time is sufficient. Most recently Chan [1995] obtained an algorithm based on the gift-wrapping method that runs in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor k/2 \rfloor+1)} \log^{O(1)} n)$ time. Note that the algorithm is optimal when $k = 2, 3$. In particular, it is optimal when $\mathcal{H} = o(n^{1-\epsilon})$ for some $0 < \epsilon < 1$.

We conclude this subsection with the following theorem [Chan 1995].

Theorem 11.6 *The convex hull of a set S of n points in $\Re^k$ can be computed in $O(n \log \mathcal{H})$ time for $k = 2$ or $k = 3$, and in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor k/2 \rfloor+1)} \log^{O(1)} n)$ time for $k > 3$, where $\mathcal{H}$ is the number of i-faces, $i = 0, 1, \ldots, k-1$.*

11.3.2 Proximity

In this subsection we address proximity related problems.

11.3.2.1 Closest Pair

Consider a set S of n points in $\Re^k$. The closest pair problem is to find in S a pair of points whose distance is the minimum, i.e., find p_i and p_j, such that $d(p_i, p_j) = \min_{k \neq l}\{d(p_k, p_l)$, for all points $p_k, p_l \in S\}$, where $d(a, b)$ denotes the Euclidean distance between a and b. (The subsequent result holds for any distance metric in Minkowski's norm.) The brute force method takes $O(d \cdot n^2)$ time by computing all $O(n^2)$ interpoint distances and taking the minimum; the pair that gives the minimum distance is the closest pair.

---

*If the points of S are not given a priori, the algorithm can be made *on line* by adding an extra step of checking if the newly added point is internal or external to the current convex hull. If internal, just discard it.

In one dimension, the problem can be solved by sorting these points and then scanning them in order, as the two closest points must occur consecutively. And this problem has a lower bound of $\Omega(n \log n)$ even in one dimension following from a linear time transformation from the *element uniqueness problem*. See Preparata and Shamos [1985].

But sorting is not applicable for dimension $k > 1$. Indeed this problem can be solved in optimal time $O(n \log n)$ by using the divide-and-conquer approach as follows. Let us first consider the case when $k = 2$. Consider a vertical cutting line λ that divides S into S_1 and S_2 such that $|S_1| = |S_2| = n/2$. Let δ_i be the minimum distance defined by points in $S_i, i = 1, 2$. Observe that the minimum distance defined by points in S can be either δ_1, δ_2, or defined by two points, one in each set. In the former case, we are done. In the latter, these two points must lie in the vertical strip of width $\delta = \min\{\delta_1, \delta_2\}$ on each side of the cutting line λ. The problem now reduces to that of finding the closest pair between points in S_1 and S_2 that lie inside the strip of width 2δ. This subproblem has a special property, known as the *sparsity* condition, i.e., the number of points in a box* of length 2δ is bounded by a constant $c = 4 \cdot 3^{k-1}$, because in each set S_i, there exists no point that lies in the interior of the δ-ball centered at each point in $S_i, i = 1, 2$ [Preparata and Shamos 1985]. It is this sparsity condition that enables us to solve the bichromatic closest pair problem (cf. the following subsection for more information) in $O(n)$ time. Let $\overline{S}_i \subseteq S_i$ denote the set of points that lies in the vertical strip. In two dimensions, the sparsity condition ensures that for each point in $\overline{S}_1$ the number of candidate points in $\overline{S}_2$ for the closest pair is at most 6. We therefore can scan these points $\overline{S}_1 \cup \overline{S}_2$ in order along the cutting line λ and compute the distance between each point scanned and its six candidate points. The pair that gives the minimum distance δ_3 is the bichromatic closest pair. The minimum distance of all pairs of points in S is then equal to $\delta_S = \min\{\delta_1, \delta_2, \delta_3\}$.

Since the merge step takes linear time, the entire algorithm takes $O(n \log n)$ time. This idea generalizes to higher dimensions, except that to ensure the sparsity condition the cutting hyperplane should be appropriately chosen to obtain an $O(n \log n)$ algorithm [Preparata and Shamos 1985].

11.3.2.2 Bichromatic Closest Pair

Given two sets of *red* and *blue* points, denoted R and B, respectively, find two points, one in R and the other in B, that are closest among all such mutual pairs.

The special case when the two sets satisfy the sparsity condition defined previously can be solved in $O(n \log n)$ time, where $n = |R| + |B|$. In fact a more general problem, known as *fixed radius all nearest-neighbor problem in a sparse set* [Bentley 1980, Preparata and Shamos 1985], i.e., given a set M of points in $\Re^k$ that satisfies the sparsity condition, find all pairs of points whose distance is less than a given parameter δ, can be solved in $O(|M| \log |M|)$ time [Preparata and Shamos 1985]. The bichromatic closest pair problem in general, however, seems quite difficult. Agarwal et al. [1991] gave an $O(n^{2(1-1/(\lceil k/2 \rceil+1))+\epsilon})$ time algorithm and a randomized algorithm with an expected running time of $O(n^{4/3} \log^c n)$ for some constant c. Chazelle et al. [1993] gave an $O(n^{2(1-1/(\lfloor k/2 \rfloor+1))+\epsilon})$ time algorithm for the bichromatic farthest pair problem, which can be used to find the diameter of a set S of points by setting $R = B = S$.

A lower bound of $\Omega(n \log n)$ for the bichromatic closest pair problem can be established. (See e.g., Preparata and Shamos [1985].) However, when the two sets are given as two simple polygons, the bichromatic closest pair problem can be solved relatively easily. Two problems can be defined. One is the *closest visible vertex pair* problem, and the other is the *separation problem*. In the former, one looks for a red–blue pair of vertices that are visible to each other and are the closest; in the latter, one looks for two boundary points that have the shortest distance. Both the closest visible vertex pair problem and the separation problem can be solved in linear time [Amato 1994, 1995]. But if both polygons are convex, the separation problem can be solved in $O(\log n)$ time [Chazelle and Dobkin 1987, Edelsbrunner 1985].

Additional references about different variations of closest pair problems can be found in Bespamyatnikh [1995], Callahan and Kosaraju [1995], Kapoor and Smid [1996], Schwartz et al. [1994], and Smid [1992].

*A box is also known as a hypercube.

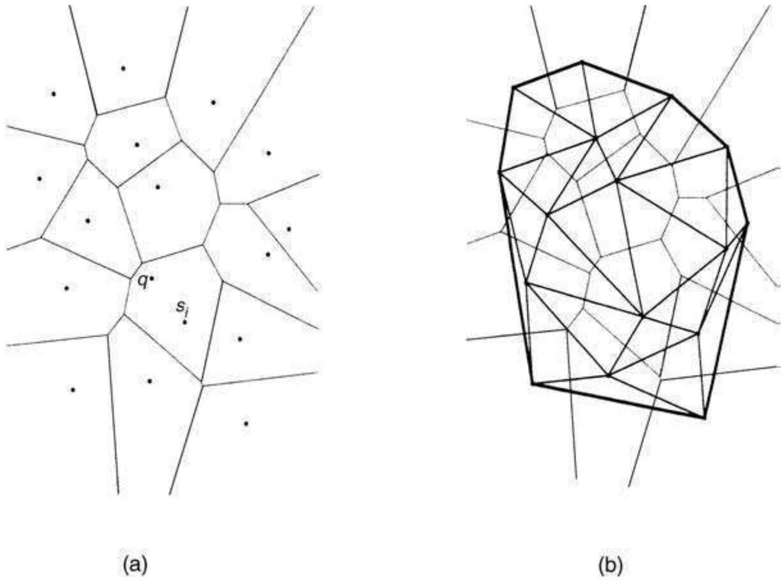


FIGURE 11.7 The Voronoi diagram of a set of 16 points in the plane.

11.3.2.3 Voronoi Diagrams

The Voronoi diagram $\mathcal{V}(S)$ of a set S of points, called *sites*, $S = \{s_1, s_2, \ldots, s_n\}$ in $\Re^k$ is a partition of $\Re^k$ into Voronoi cells $V(s_i)$, $i = 1, 2, \ldots, n$, such that each cell contains points that are closer to site s_i than to any other site s_j, $j \neq i$, i.e.,

$$V(s_i) = \{x \in \Re^k \mid d(x, s_i) \leq d(x, s_j) \forall s_j \in \Re^k, j \neq i\}$$

Figure 11.7a shows the Voronoi diagram of 16 point sites in two dimensions. Figure 11.7b shows the straight-line dual graph of the Voronoi diagram, which is called the Delaunay triangulation.

In two dimensions, $\mathcal{V}(S)$ is a planar graph and is of size linear in $|S|$. In dimensions $k \geq 2$, the total number of d-faces of dimensions $d = 0, 1, \ldots, k - 1$, in $\mathcal{V}(S)$ is $O(n^{\lceil d/2 \rceil})$.

11.3.2.3.1 Construction of Voronoi Diagram in Two Dimensions

The Voronoi diagram possesses many properties that are proximity related. For instance, the closest pair problem for S can be solved in linear time after the Voronoi diagram has been computed. Because this pair of points must be adjacent in the Delaunay triangulation, all one has to do is examine all adjacent pairs of points and report the pair with the smallest distance. A divide-and-conquer algorithm to compute the Voronoi diagram of a set of points in the Euclidean plane was first given by Shamos and Hoey and generalized by Lee to L_p-metric for all $1 \leq p \leq \infty$ [Preparata and Shamos 1985]. A *plane-sweep* technique for constructing the diagram is proposed by Fortune [1987] that runs in $O(n \log n)$ time. There is a rich body of literature concerning the Voronoi diagram. The interested reader is referred to a recent survey by Fortune in Du and Hwang [1992, pp. 192–234].

Although $\Omega(n \log n)$ is the lower bound for computing the Voronoi diagram for an arbitrary set of n sites, this lower bound does not apply to special cases, e.g., when the sites are on the vertices of a convex polygon. In fact the Voronoi diagram of a convex polygon can be computed in linear time [Aggarwal et al. 1989]. This demonstrates further that an additional property of the input is to help reduce the complexity of the problem.

11.3.2.3.2 Construction of Voronoi Diagrams in Higher Dimensions

The Voronoi diagrams in $\Re^k$ are related to the convex hulls $\Re^{k+1}$ via a geometric transformation similar to duality discussed earlier in the subsection on geometric duality. Consider a set of n sites in $\Re^k$, which is the hyperplane $\mathcal{H}^0$ in $\Re^{k+1}$ such that $x_{k+1} = 0$, and a paraboloid $\mathcal{P}$ in $\Re^{k+1}$ represented as $x_{k+1} = x_1^2 + x_2^2 + \cdots + x_k^2$. Each site $s_i = (\mu_1, \mu_2, \ldots, \mu_k)$ is transformed into a hyperplane $\mathcal{H}(s_i)$ in $\Re^{k+1}$ denoted as

$$x_{k+1} = 2\sum_{j=1}^{k} \mu_j x_j - \left(\sum_{j=1}^{k} \mu_j^2\right)$$

That is, $\mathcal{H}(s_i)$ is tangent to the paraboloid $\mathcal{P}$ at a point $\mathcal{P}(s_i) = (\mu_1, \mu_2, \ldots, \mu_k, \mu_1^2 + \mu_2^2 + \cdots + \mu_k^2)$, which is just the vertical projection of site s_i onto the paraboloid $\mathcal{P}$. The half-space defined by $\mathcal{H}(s_i)$ and containing the paraboloid $\mathcal{P}$ is denoted as $\mathcal{H}^+(s_i)$. The intersection of all half-spaces, $\bigcap_{i=1}^{n} \mathcal{H}^+(s_i)$ is a convex body, and the boundary of the convex body is denoted $CH(\mathcal{H}(S))$. Any point $p \in \Re^k$ lies in the Voronoi cell $V(s_i)$ if the vertical projection of p onto $CH(\mathcal{H}(S))$ is contained in $\mathcal{H}(s_i)$. In other words, every κ-face of $CH(\mathcal{H}(S))$ has a vertical projection on the hyperplane $\mathcal{H}^0$ equal to the κ-face of the Voronoi diagram of S in $\mathcal{H}^0$.

We thus obtain the result which follows from Theorem 11.6 [Edelsbrunner 1987].

Theorem 11.7 *The Voronoi diagram of a set S of n points in $\Re^k$, $k \geq 3$, can be computed in $O(CH_{RH}(n))$ time and $O(n^{\lceil k/2 \rceil})$ space, where $CH_\ell(n)$ denotes the time for constructing the convex hull of n points in $\Re^\ell$.*

For more results concerning the Voronoi diagrams in higher dimensions and duality transformation see Aurenhammer [1990].

11.3.2.4 Farthest-Neighbor Voronoi Diagram

The Voronoi diagram defined in the preceding subsection is also known as the nearest-neighbor Voronoi diagram. A variation of this partitioning concept is a partition of the space into cells, each of which is associated with a site, which contains all points that are farther from the site than from any other site. This diagram is called the *farthest-neighbor* Voronoi diagram. Unlike the nearest-neighbor Voronoi diagram, only a subset of sites have a Voronoi cell associated with them. Those sites that have a nonempty Voronoi cell are those that lie on the convex hull of S. A similar partitioning of the space is known as the order κ-nearest-neighbor Voronoi diagram, in which each Voronoi cell is associated with a subset of κ sites in S for some fixed integer κ such that these κ sites are the closest among all other sites. For $\kappa = 1$ we have the nearest-neighbor Voronoi diagram, and for $\kappa = n - 1$ we have the farthest-neighbor Voronoi diagram. The higher order Voronoi diagrams in k-dimensions are related to the levels of hyperplane arrangements in $k + 1$ dimensions using the paraboloid transformation [Edelsbrunner 1987].

Because the farthest-neighbor Voronoi diagram is related to the convex hull of the set of sites, one can use the marriage-before-conquest paradigm of Kirkpatrick and Seidel [1986] to compute the farthest-neighbor Voronoi diagram of S in two dimensions in time $O(n \log \mathcal{H})$, where $\mathcal{H}$ is the number of sites on the convex hull.

11.3.2.5 Weighted Voronoi Diagrams

When the sites are associated with weights such that the distance function from a point to the sites is weighted, the structure of the Voronoi diagram can be drastically different than the unweighted case.

11.3.2.5.1 Power Diagrams

Suppose each site s in $\Re^k$ is associated with a nonnegative weight, w_s. For an arbitrary point p in $\Re^k$ the weighted distance from p to s is defined as

$$\delta(s, p) = d(s, p)^2 - w_s^2$$

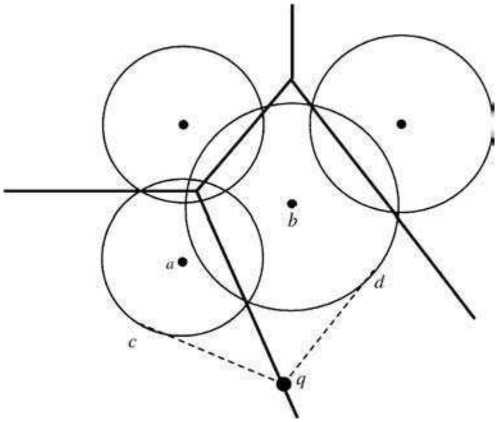


FIGURE 11.8 The power diagram in two dimensions; solid lines are equidistant to two sites.

If w_s is positive, and if $d(s, p) \geq w_s$, then $\sqrt{\delta(s, p)}$ is the length of the tangent of p to the ball $b(s)$ of radius w_s and centered at s. Here $\delta(s, p)$ is also called the *power of p* with respect to the ball $b(s)$. The locus of points p equidistant from two sites $s \neq t$ of equal weight will be a hyperplane called the *chordale* of s and t. See Figure 11.8. Point q is equidistant to sites a and b, and the distance is the length of the tangent line $\overline{q, c} = \overline{q, d}$.

The power diagram of two dimensions can be used to compute the contour of the union of n disks and the connected components of n disks in $O(n \log n)$ time, and in higher dimensions it can be used to compute the union or intersection of n axis-parallel cones in $\Re^k$ with apices in a common hyperplane in time $O(CH_{k+1}(n))$, the multiplicative weighted nearest-neighbor Voronoi diagram (defined subsequently) for n points in $\Re^k$ in time $O(CH_{k+2}(n))$, and the Voronoi diagrams for n spheres in $\Re^k$ in time $O(CH_{k+2}(n))$, where $CH_\ell(n)$ denotes the time for constructing the convex hull of n points in $\Re^\ell$ [Aurenhammer 1987]. For the best time bound for $CH_\ell(n)$ consult the subsection on convex hulls.

11.3.2.5.2 Multiplicative-Weighted Voronoi Diagrams

Each site $s \in \Re^k$ has a positive weight w_s, and the distance from a point p to s is defined as

$$\delta_{\text{multi}-w}(s, p) = d(p, s)/w_s$$

In two dimensions, the locus of points equidistant to two sites $s \neq t$ is a circle, if $w_s \neq w_t$, and a perpendicular bisector of line segment $\overline{s, t}$, if $w_s = w_t$. Each cell associated with a site s consists of all points closer to s than to any other site and may be disconnected. In the worst case the nearest-neighbor Voronoi diagram of a set S of n points in two dimensions can have an $O(n^2)$ regions and can be found in $O(n^2)$ time. In one dimension, the diagram can be computed optimally in $O(n \log n)$ time. However, the farthest-neighbor multiplicative-weighted Voronoi diagram has a very different characteristic. Each Voronoi cell associated with a site remains connected, and the size of the diagram is still linear in the number of sites. An $O(n \log^2 n)$ time algorithm for constructing such a diagram is given in Lee and Wu [1993]. See Schaudt and Drysdale [1991] for more applications of the diagram.

11.3.2.5.3 Additive-Weighted Voronoi Diagrams

The distance of a point p to a site s of a weight w_s is defined as

$$\delta_{\text{add}-w}(s, p) = d(p, s) - w_s$$

In two dimensions, the locus of points equidistant to two sites $s \neq t$ is a branch of a hyperbola, if $w_s \neq w_t$, and a perpendicular bisector of line segment $\overline{s, t}$ if $w_s = w_t$. The Voronoi diagram has properties

similar to the ordinary unweighted diagram. For example, each cell is still connected and the size of the diagram is linear. If the weights are positive, the diagram is the same as the Voronoi diagram of a set of spheres centered at site s and of radius w_s, in two dimensions this diagram for n disks can be computed in $O(n \log^2 n)$ time [Lee and Drysdale 1981, Sharir 1985], and in $k \geq 3$ one can use the notion of power diagram to compute the diagram [Aurenhammer 1987].

11.3.2.6 Other Generalizations

The sites mentioned so far are point sites. They can be of different shapes. For instance, they can be line segments, disks, or polygonal objects. The metric used can also be a convex distance function or other norms. See Alt and Schwarzkopf [1995], Boissonnat et al. [1995], Klein [1989], and Yap [1987a] for more information.

11.3.3 Point Location

Point location is yet another fundamental problem in computational geometry. Given a planar subdivision and a query point, one would like to find which region contains the point in question.

In this context, we are mostly interested in fast response time to answer repeated queries to a fixed database. An earlier approach is based on the *slab method* [Preparata and Shamos 1985], in which parallel lines are drawn through each vertex, thus partitioning the plane into parallel slabs. Each parallel slab is further divided into subregions by the edges of the subdivision that can be ordered. Any given point can thus be located by two binary searches: one to locate the slab containing the point among the $n + 1$ horizontal slabs, followed by another to locate the region defined by a pair of consecutive edges that are ordered from left to right. This requires preprocessing of the planar subdivision, and setting up suitable search tree structures for the slabs and the edges crossing each slab. We use a three-tuple, $(P(n), S(n), Q(n)) =$ (preprocessing time, space requirement, query time) to denote the performance of the search strategy (cf. section on dynamization). The slab method gives an $(O(n^2), O(n^2), O(\log n))$ algorithm. Because preprocessing time is only performed once, the time requirement is not as critical as the space requirement. The primary goal of the query processing problems is to minimize the query time and the space required.

Lee and Preparata first proposed a *chain decomposition* method to decompose a monotone planar subdivision with n points into a collection of $m \leq n$ monotone chains organized in a complete binary tree [Preparata and Shamos 1985]. Each node in the binary tree is associated with a monotone chain of at most n edges, ordered in the y-coordinate. Between two adjacent chains, there are a number of disjoint regions. Each query point is compared with the node, hence the associated chain, to decide on which side of the chain the query point lies. Each chain comparison takes $O(\log n)$ time, and the total number of nodes visited is $O(\log m)$. The search on the binary tree will lead to two adjacent chains and hence identify a region that contains the point. Thus, the query time is $O(\log m \log n) = O(\log^2 n)$. Unlike the slab method in which each edge may be stored as many as $O(n)$ times, resulting in $O(n^2)$ space, it can be shown that each edge in the planar subdivision, with an appropriate chain assignment scheme, is stored only once. Thus, the space requirement is $O(n)$. The chain decomposition scheme gives rise to an $(O(n \log n), O(n), O(\log^2 n))$ algorithm. The binary search on the chains is not efficient enough. Recall that after each *chain comparison*, we will move down the binary search tree to perform the next chain comparison and start over another binary search on the y-coordinate to find an edge of the chain, against which a comparison is made to decide if the point lies to the left or right of the chain. A more efficient scheme is to perform a binary search of the y-coordinate at the root node and to spend only $O(1)$ time per node as we go down the chain tree, shaving off an $O(\log n)$ factor from the query time [Edelsbrunner et al. 1986]. This scheme is similar to the ones adopted by Chazelle and Guibas [1986] in a fractional cascading search paradigm and by Willard [1985] in his range tree search method. With the linear time algorithm for triangulating a simple polygon due to Chazelle [1991] (cf. subsequent subsection on triangulation) we conclude with the following optimal search structure for planar point location.

Theorem 11.8 *Given a planar subdivision of n vertices, one can preprocess the subdivision in linear time and space such that each point location query can be answered in* $O(\log n)$ *time.*

The point location problem in arrangements of hyperplanes is also of significant interest. See, e.g., Chazelle and Friedman [1990]. Dynamic versions of the point location problem have also been investigated. See Chiang and Tamassia [1992] for a survey of dynamic computational geometry.

11.3.4 Motion Planning: Path Finding Problems

The problem is mostly cast in the following setting. Given are a set of obstacles O, an object, called *robot*, and an initial and final position, called source and destination, respectively. We wish to find a path for the robot to move from the source to the destination, avoiding all of the obstacles. This problem arises in several contexts. For instance, in robotics this is referred to as the *piano movers' problem* [Yap 1987b] or *collision avoidance* problem, and in VLSI routing this is the *wiring* problem for 2-terminal nets. In most applications we are searching for a collision avoidance path that has a shortest length, where the distance measure is based on the Euclidean or L_1-metric. For more information regarding motion planning see, e.g., Alt and Yap [1990] and Yap [1987b].

11.3.4.1 Path Finding in Two Dimensions

In two dimensions, the Euclidean shortest path problem in which the robot is a point and the obstacles are simple polygons, is well studied. A most fundamental approach is by using the notion of *visibility graph*. Because the shortest path must make turns at polygonal vertices, it is sufficient to construct a graph whose vertices are the vertices of the polygonal obstacles and the source and destination and whose edges are determined by vertices that are mutually *visible*, i.e., the segment connecting the two vertices does not intersect the interior of any obstacle. Once the visibility graph is constructed with edge weight equal to the Euclidean distance between the two vertices, one can then apply Dijkstra's shortest path algorithms [Preparata and Shamos 1985] to find a shortest path between the source and destination. The Euclidean shortest path between two points is referred to as the *geodesic path* and the distance as the *geodesic distance*. The computation of the visibility graph is the dominating factor for the complexity of any visibility graph-based shortest path algorithm. Research results aiming at more efficient algorithms for computing the visibility graph and for computing the geodesic path in time proportional to the size of the graph have been obtained. Ghosh and Mount [1991] gave an output-sensitive algorithm that runs in $O(E + n \log n)$ time for computing the visibility graph, where E denotes the number of edges in the graph.

Mitchell [1993] used the so-called *continuous Dijkstra* wave front approach to the problem for the general polygonal domain of n obstacle vertices and obtained an $O(n^{5/3+\epsilon})$ time algorithm. He constructed a *shortest path map* that partitions the plane into regions such that all points q that lie in the same region have the same vertex sequence in the shortest path from the given source to q. The shortest path map takes $O(n)$ space and enables us to perform shortest path queries, i.e., find a shortest path from the given source to any query points, in $O(\log n)$ time. Hershberger and Suri [1993] on the other hand, used a plane subdivision approach and presented an $O(n \log^2 n)$-time and $O(n \log n)$-space algorithm to compute the shortest path map of a given source point. They later improved the time bound to $O(n \log n)$. If the source-destination path is confined in a simple polygon with n vertices, the shortest path can be found in $O(n)$ time [Preparata and Shamos 1985].

In the context of VLSI routing one is mostly interested in rectilinear paths (L_1-metric) whose edges are either horizontal or vertical. As the paths are restricted to be rectilinear, the shortest path problem can be solved more easily. Lee et al. [1996] gave a survey on this topic.

In a two-layer VLSI routing model, the number of segments in a rectilinear path reflects the number of *vias*, where the wire segments change layers, which is a factor that governs the fabrication cost. In robotics, a straight-line motion is not as costly as making turns. Thus, the number of segments (or *turns*) has also

become an objective function. This motivates the study of the problem of finding a path with the smallest number of segments, called the *minimum link path problem* [Mitchell et al. 1992, Suri 1990].

These two cost measures, length and number of links, are in conflict with each other. That is, a shortest path may have far too many links, whereas a minimum link path may be arbitrarily long compared with a shortest path. Instead of optimizing both measures *simultaneously*, one can seek a path that either optimizes a linear function of both length and the number of links or optimizes them in a lexicographical order. For example, we optimize the length first, and then the number of links, i.e., among those paths that have the same shortest length, find one whose number of links is the smallest, and vice versa.

A generalization of the collision-avoidance problem is to allow collision with a cost. Suppose each obstacle has a weight, which represents the cost if the obstacle is *penetrated*. Mitchell and Papadimitriou [1991] first studied the weighted region shortest path problem. Lee et al. [1991] studied a similar problem in the rectilinear case. Another generalization is to include in the set of obstacles some subset $F \subset O$ of obstacles, whose vertices are *forbidden* for the solution path to make turns. Of course, when the weight of obstacles is set to be ∞, or the forbidden set $F = \emptyset$, these generalizations reduce to the ordinary collision-avoidance problem.

11.3.4.2 Path Finding in Three Dimensions

The Euclidean shortest path problem between two points in a three-dimensional polyhedral environment turns out to be much harder than its two-dimensional counterpart. Consider a convex polyhedron P with n vertices in three dimensions and two points s, d on the surface of P. A shortest path from s to d on the surface will cross a sequence of edges, denoted $\xi(s, d)$. Here $\xi(s, d)$ is called the *shortest path edge sequence* induced by s and d and consists of distinct edges. If the edge sequence is known, the shortest path between s and d can be computed by a planar unfolding procedure so that these faces crossed by the path lie in a common plane and the path becomes a straight-line segment.

Mitchell et al. [1987] gave an $O(n^2 \log n)$ algorithm for finding a shortest path between s and d even if the polyhedron may not be convex. If s and d lie on the surface of two different polyhedra, Sharir [1987] gave an $O(N^{O(k)})$ algorithm, where N denotes the total number of vertices of k obstacles. In general, the problem of determining the shortest path edge sequence of a path between two points among k polyhedra is NP-hard [Canny and Reif 1987].

11.3.4.3 Motion Planning of Objects

In the previous sections, we discussed path planning for moving a point from the source to a destination in the presence of polygonal or polyhedral obstacles. We now briefly describe the problem of moving a polygonal or polyhedral object from an initial position to a final position subject to translational and/or rotational motions.

Consider a set of k convex polyhedral obstacles, $O_1, O_2, \ldots, O_k$, and a convex polyhedral robot, R in three dimensions. The motion planning problem is often solved by using the so-called *configuration space*, denoted $\mathcal{C}$, which is the space of parametric representations of possible robot placements [Lozano-Pérez 1983]. The free placement (FP) is the subspace of $\mathcal{C}$ of points at which the robot does not intersect the interior of any obstacle. For instance, if only translations of R are allowed, the free configuration space will be the union of the Minkowski sums $M_i = O_i \oplus (-R) = \{a - b \mid a \in O_i, b \in R\}$ for $i = 1, 2, \ldots, k$. A *feasible* path exists if the initial placement of R and final placement belong to the same connected component of FP. The problem is to find a continuous curve connecting the initial and final positions in FP. The combinatorial complexity, i.e., the number of vertices, edges, and faces on the boundary of FP, largely influences the efficiency of any $\mathcal{C}$-based algorithm. For translational motion planning, Aronov and Sharir [1994] showed that the combinatorial complexity of FP is $O(nk \log^2 k)$, where k is the number of obstacles defined above and n is the total complexity of the Minkowski sums $M_i, 1 \le i \le k$.

Moving a ladder (represented as a line segment) among a set of polygonal obstacles of size n can be done in $O(K \log n)$ time, where K denotes the number of pairs of obstacle vertices whose distance is less than the length of the ladder and is $O(n^2)$ in general [Sifrony and Sharir 1987]. If the moving robot is

also a polygonal object, Avnaim et al. [1988] showed that $O(n^3 \log n)$ time suffices. When the obstacles are *fat** Van der Stappen and Overmars [1994] showed that the two preceding two-dimensional motion planning problems can be solved in $O(n \log n)$ time, and in three dimensions the problem can be solved in $O(n^2 \log n)$ time, if the obstacles are ℓ-fat for some positive constant ℓ.

11.3.5 Geometric Optimization

The geometric optimization problems arise in operations research, pattern recognition, and other engineering disciplines. We list some representative problems.

11.3.5.1 Minimum Cost Spanning Trees

The minimum (cost) spanning tree MST of an undirected, weighted graph $G(V, E)$, in which each edge has a nonnegative weight, is a well-studied problem in graph theory and can be solved in $O(|E| \log |V|)$ time [Preparata and Shamos 1985]. When cast in the Euclidean or other L_p-metric plane in which the input consists of a set S of n points, the complexity of this problem becomes different. Instead of constructing a *complete* graph whose edge weight is defined by the distance between its two endpoints, from which to extract an MST, a sparse graph, known as the *Delaunay triangulation* of the point set, is computed. It can be shown that the MST of S is a subgraph of the Delaunay triangulation. Because the MST of a planar graph can be found in linear time [Preparata and Shamos 1985], the problem can be solved in $O(n \log n)$ time. In fact, this is asymptotically optimal, as the closest pair of the set of points must define an edge in the MST, and the closest pair problem is known to have an $\Omega(n \log n)$ lower bound, as mentioned previously.

This problem in three or more dimensions can be solved in subquadratic time. For instance, in three dimensions $O((n \log n)^{1.5})$ time is sufficient [Chazelle 1985] and in $k \geq 3$ dimensions $O(n^{2(1-1/(\lceil k/2 \rceil+1))+\epsilon})$ time suffices [Agarwal et al. 1991].

11.3.5.2 Minimum Diameter Spanning Tree

The minimum *diameter* spanning tree (MDST) of an undirected, weighted graph $G(V, E)$ is a spanning tree such that the total weight of the longest path in the tree is minimum. This arises in applications to communication networks where a tree is sought such that the maximum delay, instead of the total cost, is to be minimized. A graph-theoretic approach yields a solution in $O(|E||V| \log |V|)$ time [Handler and Mirchandani 1979]. Ho et al. [1991] showed that by the triangle inequality there exists an MDST such that the longest path in the tree consists of no more than *three* segments. Based on this an $O(n^3)$ time algorithm was obtained.

Theorem 11.9 *Given a set S of n points, the minimum diameter spanning tree for S can be found in $\theta(n^3)$ time and $O(n)$ space.*

We remark that the problem of finding a spanning tree whose total cost and the diameter are both bounded is NP-complete [Ho et al. 1991]. A similar problem that arises in VLSI clock tree routing is to find a tree from a source to multiple sinks such that every source-to-sink path is the shortest and the total wire length is to be minimized. This problem still is not known to be solvable in polynomial time or NP-hard. Recently, we have shown that the problem of finding a minimum spanning tree such that the longest source-to-sink path is bounded by a given parameter is NP-complete [Seo and Lee 1995].

11.3.5.3 Minimum Enclosing Circle Problem

Given a set S of points, the problem is to find the smallest disk enclosing the set. This problem is also known as the (unweighted) one-center problem. That is, find a center such that the maximum distance

*An object $O \subseteq R^k$ is said to be *ℓ-fat* if for all hyperspheres S centered inside O and not fully containing O we have $\ell \cdot$ *volume* $(O \cap S) \geq$ *volume*(S).

from the center to the points in S is minimized. More formally, we need to find the center $c \in \Re^2$ such that $\max_{p_j \in S} d(c, p_j)$ is minimized. The weighted one-center problem, in which the distance function $d(c, p_j)$ is multiplied by the weight w_j, is a well-known minimax problem, also known as the *emergency center problem* in operations research. In two dimensions, the one-center problem can be solved in $O(n)$ time [Dyer 1986, Megiddo 1983b]. The minimum enclosing ball problem in higher dimensions is also solved by using a linear programming technique [Megiddo 1983b, 1984].

11.3.5.4 Largest Empty Circle Problem

This problem, in contrast to the minimum enclosing circle problem, is to find a circle centered in the interior of the convex hull of the set S of points that does not contain any given point and the radius of the circle is to be maximized. This is mathematically formalized as a maximin problem; the minimum distance from the center to the set is maximized. The weighted version is also known as the *obnoxious center* problem in facility location. An $O(n \log n)$ time solution for the unweighted version can be found in [Preparata and Shamos 1985].

11.3.5.5 Minimum Annulus Covering Problem

The *minimum annulus covering problem* is defined as follows. Given a set of S of n points find an annulus (defined by two concentric circles) whose center lies internal to the convex hull of S such that the *width* of the annulus is minimized. The problem arises in mechanical part design. To measure whether a circular part is *round*, an American National Standards Institute (ANSI) standard is to use the width of an annulus covering the set of points obtained from a number of measurements. This is known as the *roundness* problem [Le and Lee 1991]. It can be shown that the center of the annulus is either at a vertex of the nearest-neighbor Voronoi diagram, a vertex of the farthest-neighbor Voronoi diagram, or at the intersection of these two diagrams [Le and Lee 1991]. If the input is defined by a simple polygon P with n vertices, and the problem is to find a minimum-width annulus that contains the boundary of P, the problem can be solved in $O(n \log n + k)$, where k denotes the number of intersection points of the *medial axis* of the simple polygon and the boundary of P [Le and Lee 1991]. When the polygon is known to be convex, a linear time is sufficient [Swanson et al. 1995]. If the center of the smallest annulus of a point set can be arbitrarily placed, the center may lie at infinity and the annulus degenerates to a pair of parallel lines enclosing the set of points. This problem is different from the problem of finding the width of a set, which is to find a pair of parallel lines enclosing the set such that the distance between them is minimized. The width of a set of n points can be found in $O(n \log n)$ time, which is optimal [Lee and Wu 1986]. In three dimensions the *width* of a set is also used as a measure for flatness of a *plate—flatness* problem. Houle and Toussaint [1988] gave an $O(n^2)$ time algorithm, and Chazelle et al. [1993] improved it to $O(n^{8/5+\epsilon})$.

11.3.6 Decomposition

Polygon decomposition arises in pattern recognition in which recognition of a shape is facilitated by first decomposing it into simpler parts, called *primitives*, and comparing them to templates previously stored in a library via some similarity measure. The primitives are often convex, with the simplest being the shape of a triangle.

We consider two types of decomposition, *partition* and *covering*. In the former type, the components are pairwise disjoint except they may have some boundary edges in common. In the latter type, the components may overlap. A minimum decomposition is one such that the number of components is minimized. Sometimes additional points, called *Steiner points*, may be introduced to obtain a minimum decomposition. Unless otherwise specified, we assume that no Steiner points are used.

11.3.6.1 Triangulation

Triangulating a simple polygon or, in general, triangulating a planar straight-line graph, is a process of introducing noncrossing edges so that each face is a triangle. It is also a fundamental problem in computer graphics, geographical information systems, and finite-element methods.

Let us begin with the problem of triangulating a simple polygon with n vertices. It is obvious that for a simple polygon with n edges, one needs to introduce at most $n-3$ diagonals to triangulate the interior into $n-2$ triangles. This problem has been studied very extensively. A pioneering work is due to Garey et al., which gave an $O(n \log n)$ algorithm and a linear algorithm if the polygon is monotone [O'Rourke 1994, Preparata and Shamos 1985]. A breakthrough linear time triangulation result of Chazelle [1991] settled the long-standing open problem. As a result of this linear triangulation algorithm, a number of problems can be solved in linear time, for example, the simplicity test, defined subsequently, and many other shortest path problems inside a simple polygon [Guibas and Hershberger 1989]. Note that if the polygons have holes, the problem of triangulating the interior requires $\Omega(n \log n)$ time [Asano et al. 1986].

Sometimes we want to look for *quality* triangulation instead of just an arbitrary one. For instance, triangles with large or small angles are not desirable. It is well known that the Delaunay triangulation of points in general position is unique, and it will maximize the minimum angle. In fact, the characteristic angle vector* of the Delaunay triangulation of a set of points is *lexicographically maximum* [Lee 1978]. The notion of Delaunay triangulation of a set of points can be generalized to a planar straight-line graph $G(V, E)$. That is, we would like to have G as a subgraph of a triangulation $G'(V, E')$, $E \subseteq E'$, such that each triangle satisfies the *empty circumcircle* property; no vertex visible from the vertices of a triangle is contained in the interior of the circle. This *generalized* Delaunay triangulation was first introduced by Lee [1978] and an $O(n^2)$ (respectively, $O(n \log n)$) algorithm for constructing the generalized triangulation of a planar graph (respectively, a simple polygon) with n vertices was given in Lee and Lin [1986b]. Chew [1989] later improved the result and gave an $O(n \log n)$ time algorithm using divide-and-conquer. Triangulations that minimize the maximum angle or maximum edge length were also studied. But if constraints on the measure of the triangles, for instance, each triangle in the triangulation must be nonobtuse, then Steiner points must be introduced. See Bern and Eppstein (in Du and Hwang [1992, pp. 23–90]) for a survey of different criteria of triangulations and discussions of triangulations in two and three dimensions.

The problem of triangulating a set P of points in $\Re^k$, $k \geq 3$, is less studied. In this case, the convex hull of P is to be partitioned into $\mathcal{F}$ nonoverlapping simplices, the vertices of which are points in P. A simplex in k-dimensions consists of exactly $k+1$ points, all of which are extreme points. Avis and ElGindy [1987] gave an $O(k^4 n \log_{1+1/k} n)$ time algorithm for triangulating a simplicial set of n points in $\Re^k$. In $\Re^3$ an $O(n \log n + \mathcal{F})$ time algorithm was presented and $\mathcal{F}$ is shown to be linear if no three points are collinear and at most $O(n^2)$ otherwise. See Du and Hwang [1992] for more references on three-dimensional triangulations and Delaunay triangulations in higher dimensions.

11.3.6.2 Other Decompositions

Partitioning a simple polygon into shapes such as convex polygons, star-shaped polygons, spiral polygons, monotone polygons, etc., has also been investigated [Toussaint 1985]. A linear time algorithm for partitioning a polygon into star-shaped polygons was given by Avis and Toussaint [1981] after the polygon has been triangulated. This algorithm provided a very simple proof of the traditional art gallery problem originally posed by Klee, i.e., $\lfloor n/3 \rfloor$ vertex guards are always sufficient to see the entire region of a simple polygon with n vertices. But if a minimum partition is desired, Keil [1985] gave an $O(n^5 N^2 \log n)$ time, where N denotes the number of reflex vertices. However, the problem of *covering* a simple polygon with a minimum number of star-shaped parts is NP-hard [Lee and Lin 1986a]. The problem of partitioning a polygon into a minimum number of convex parts can be solved in $O(N^2 n \log n)$ time [Keil 1985]. The minimum covering problem by star-shaped polygons for rectilinear polygons is still open. For variations and results of art gallery problems the reader is referred to O'Rourke [1987] and Shermer [1992]. Polynomial time algorithms for computing the minimum partition of a simple polygon into simpler parts while allowing Steiner points can be found in Asano et al. [1986] and Toussaint [1985].

*The characteristic angle vector of a triangulation is a vector of minimum angles of each triangle arranged in nondescending order. For a given point set, the number of triangles is the same for all triangulations, and therefore each of them is associated with a characteristic angle vector.

The minimum partition or covering problem for simple polygons becomes NP-hard when the polygons are allowed to have *holes* [Keil 1985, O'Rourke and Supowit 1983]. Asano et al. [1986] showed that the problem of partitioning a simple polygon with h holes into a minimum number of trapezoids with two horizontal sides can be solved in $O(n^{h+2})$ time and that the problem is NP-complete if h is part of the input. An $O(n \log n)$ time 3-approximation algorithm was presented. Imai and Asano [1986] gave an $O(n^{3/2} \log n)$ time and $O(n \log n)$ space algorithm for partitioning a rectilinear polygon with holes into a minimum number of rectangles (allowing Steiner points). The problem of covering a rectilinear polygon (without holes) with a minimum number of rectangles, however, is also NP-hard [Culberson and Reckhow 1988].

The problem of minimum partition into convex parts and the problem of determining if a nonconvex polyhedron can be partitioned into tetrahedra without introducing Steiner points are NP-hard [O'Rourke and Supowit 1983, Ruppert and Seidel 1992].

11.3.7 Intersection

This class of problems arises in architectural design, computer graphics [Dorward 1994], etc., and encompasses two types of problems, *intersection detection* and *intersection computation*.

11.3.7.1 Intersection Detection Problems

The intersection detection problem is of the form: Given a set of objects, do any two intersect? The intersection detection problem has a lower bound of $\Omega(n \log n)$ [Preparata and Shamos 1985]. The pairwise intersection detection problem is a precursor to the general intersection detection problem.

In two dimensions the problem of detecting if two polygons of r and b vertices intersect was easily solved in $O(n \log n)$ time, where $n = r + b$ using the red–blue segment intersection algorithm [Mairson and Stolfi 1988]. However, this problem can be reduced in linear time to the problem of detecting the self-intersection of a polygonal curve. The latter problem is known as the *simplicity* test and can be solved optimally in linear time by Chazelle's [1991] linear time triangulation algorithm. If the two polygons are convex, then $O(\log n)$ suffices [Chazelle and Dobkin 1987, Edelsbrunner 1985]. We remark here that, although detecting whether two convex polygons intersect can be done in logarithmic time, detecting whether the boundary of the two convex polygons intersects requires $\Omega(n)$ time [Chazelle and Dobkin 1987].

In three dimensions, detecting if two convex polyhedra intersect can be solved in linear time by using a hierarchical representation of the convex polyhedron, or by formulating it as a linear programming problem in three variables [Chazelle and Dobkin 1987, Dobkin and Kirkpatrick 1985, Dyer 1984, Megiddo 1983b].

For some applications, we would not only detect intersection but also *report* all such intersecting pairs of objects or *count* the number of intersections, which is discussed next.

11.3.7.2 Intersection Reporting/Counting Problems

One of the simplest of such intersecting reporting problems is that of *reporting* all intersecting pairs of line segments in the plane. Using the plane sweep technique, one can obtain an $O((n + \mathcal{F}) \log n)$ time, where $\mathcal{F}$ is the output size. It is not difficult to see that the lower bound for this problem is $\Omega(n \log n + \mathcal{F})$; thus the preceding algorithm is $O(\log n)$ factor from the optimal. Recently, this segment intersection reporting problem was solved optimally by Chazelle and Edelsbrunner [1992], who used several important algorithm design and data structuring techniques as well as some crucial combinatorial analysis. In contrast to this asymptotically optimal *deterministic* algorithm, a simpler randomized algorithm for this problem that takes $O(n \log n + \mathcal{F})$ time but requires only $O(n)$ space (instead of $O(n + \mathcal{F})$) was obtained [Du and Hwang 1992]. Balaban [1995] recently reported a deterministic algorithm that solves this problem optimally both in time and space.

On a separate front, the problem of finding intersecting pairs of segments from different sets was considered. This is called the *bichromatic line segment* intersection problem. Nievergelt and Preparata [1982] considered the problem of merging two planar convex subdivisions of total size n and showed that

the resulting subdivision can be computed in $O(n \log n + \mathcal{F})$ time. This result [Nievergelt and Preparata 1982] was extended in two ways. Mairson and Stolfi [1988] showed that the bichromatic line segment intersection reporting problem can be solved in $O(n \log n + \mathcal{F})$ time. Guibas and Seidel [1987] showed that merging two convex subdivisions can actually be solved in $O(n + \mathcal{F})$ time using topological plane sweep.

Most recently, Chazelle et al. [1994] used *hereditary segment trees* structure and *fractional cascading* [Chazelle and Guibas 1986] and solved both segment intersection reporting and counting problems optimally in $O(n \log n)$ time and $O(n)$ space. (The term $\mathcal{F}$ should be included for reporting.)

The *rectangle intersection reporting* problem arises in the design of VLSI circuitry, in which each rectangle is used to model a certain circuitry component. This is a well-studied classic problem and optimal algorithms ($O(n \log n + \mathcal{F})$ time) have been reported (see Lee and Preparata [1984] for references). The k-dimensional hyperrectangle intersection reporting (respectively, counting) problem can be solved in $O(n^{k-2} \log n + \mathcal{F})$ time and $O(n)$ space [respectively, in time $O(n^{k-1} \log n)$ and space $O(n^{k-2} \log n)$].

11.3.7.3 Intersection Computation

Computing the actual intersection is a basic problem, whose efficient solutions often lead to better algorithms for many other problems.

Consider the problem of computing the common intersection of half-planes discussed previously. Efficient computation of the intersection of two convex polygons is required. The intersection of two convex polygons can be solved very efficiently by plane sweep in linear time, taking advantage of the fact that the edges of the input polygons are ordered. Observe that in each vertical strip defined by two consecutive sweep lines, we only need to compute the intersection of two trapezoids, one derived from each polygon [Preparata and Shamos 1985].

The problem of intersecting two convex polyhedra was first studied by Muller and Preparata [Preparata and Shamos 1985], who gave an $O(n \log n)$ algorithm by reducing the problem to the problems of intersection detection and convex hull computation. From this one can easily derive an $O(n \log^2 n)$ algorithm for computing the common intersection of n half-spaces in three dimensions by the divide-and-conquer method. However, using geometric duality and the concept of separating plane, Preparata and Muller [Preparata and Shamos 1985] obtained an $O(n \log n)$ algorithm for this problem, which is asymptotically optimal. There appears to be a difference in the approach to solving the common intersection problem of half-spaces in two and three dimensions. In the latter, we resorted to geometric duality instead of divide-and-conquer. This *inconsistency* was later resolved. Chazelle [1992] combined the hierarchical representation of convex polyhedra, geometric duality, and other ingenious techniques to obtain a linear time algorithm for computing the intersection of two convex polyhedra. From this result several problems can be solved optimally: (1) the common intersection of half-spaces in three dimensions can now be solved by divide-and-conquer optimally, (2) the merging of two Voronoi diagrams in the plane can be done in linear time by observing the relationship between the Voronoi diagram in two dimensions and the convex hull in three dimensions (cf. subsection on Voronoi diagrams), and (3) the medial axis of a simple polygon or the Voronoi diagram of vertices of a convex polygon can be solved in linear time.

11.3.8 Geometric Searching

This class of problems is cast in the form of query answering as discussed in the subsection on dynamization. Given a collection of objects, with preprocessing allowed, one is to find objects that satisfy the queries. The problem can be static or dynamic, depending on whether the database is allowed to change over the course of query-answering sessions, and it is studied mostly in modes, *count-mode* and *report-mode*. In the former case only the number of objects satisfying the query is to be answered, whereas in the latter the actual identity of the objects is to be reported. In the report mode the query time of the algorithm consists of two components, *search time* and *output*, and expressed as $Q_A(n) = O(f(n) + \mathcal{F})$, where n denotes the size of the database, $f(n)$ a function of n, and $\mathcal{F}$ the size of output. It is obvious that algorithms that handle the report-mode queries can also handle the count-mode queries ($\mathcal{F}$ is the answer). It seems natural to expect

that the algorithms for count-mode queries would be more efficient (in terms of the order of magnitude of the space required and query time), as they need not search for the objects. However, it was argued that in the report-mode range searching, one could take advantage of the fact that since reporting takes time, the more there is to report, the *sloppier* the search can be. For example, if we were to know that the ratio $n/\mathcal{F}$ is $O(1)$, we could use a sequential search on a linear list. Chazelle in his seminal paper on filtering search capitalizes on this observation and improves the time complexity for searching for several problems [Chazelle 1986]. As indicated subsequently, the count-mode range searching problem is harder than the report-mode counterpart.

11.3.8.1 Range Searching Problems

This is a fundamental problem in database applications. We will discuss this problem and the algorithm in two dimensions. The generalization to higher dimensions is straightforward using a known technique [Bentley 1980]. Given is a set of n points in the plane, and the ranges are specified by a product $(l_1, u_1) \times (l_2, u_2)$. We would like to find points $p = (x, y)$ such that $l_1 \leq x \leq u_1$ and $l_2 \leq y \leq u_2$. Intuitively we want to find those points that lie inside a query rectangle specified by the range. This is called *orthogonal range searching*, as opposed to other kinds of range searching problems discussed subsequently. Unless otherwise specified, a range refers to an orthogonal range. We discuss the static case; as this belongs to the class of decomposable searching problems, the dynamization transformation techniques can be applied. We note that the range tree structure mentioned later can be made dynamic by using a weight-balanced tree, called a $BB(\alpha)$ tree [Mehlhorn 1984, Willard and Luecker 1985].

For count-mode queries this problem can be solved by using the locus method as follows. Divide the plane into $O(n^2)$ cells by drawing horizontal and vertical lines through each point. The answer to the query q, i.e., find the number of points dominated by q (those points whose x- and y-coordinates are both no greater than those of q) can be found by locating the cell containing q. Let it be denoted by $Dom(q)$. Thus, the answer to the count-mode range queries can be obtained by some simple arithmetic operations of $Dom(q_i)$ for the four corners of the query rectangle. We have $Q(k, n) = O(k \log n)$, $S(k, n) = P(k, n) = O(n^k)$. To reduce the space requirement at the expense of query time has been a goal of further research on this topic. Bentley [1980] introduced a data structure, called *range trees*. Using this structure the following results were obtained: for $k \geq 2$, $Q(k, n) = O(\log^{k-1} n)$, $S(k, n) = P(k, n) = O(n \log^{k-1} n)$. (See Lee and Preparata [1984] and Willard [1985] for more references.)

For report-mode queries, Chazelle [1986] showed that by using a filtering search technique the space requirement can be further reduced by a $\log \log n$ factor. In essence we use less space to allow for more objects than necessary to be found by the search mechanism, followed by a filtering process leaving out unwanted objects for output. If the range satisfies additional conditions, e.g., *grounded* in one of the coordinates, say, $l_1 = 0$, or the aspect ratio of the intervals specifying the range is fixed, then less space is needed. For instance, in two dimensions, the space required is linear (a saving of $\log n / \log \log n$ factor) for these two cases. By using the so-called functional approach to data structures Chazelle [1988] developed a *compression* scheme to encode the *downpointers* used by Willard [1985] to reduce further the space requirement. Thus in k-dimensions, $k \geq 2$, for the count-mode range queries we have $Q(k, n) = O(\log^{k-1} n)$ and $S(k, n) = O(n \log^{k-2} n)$ and for report-mode range queries $Q(k, n) = O(\log^{k-1} n + \mathcal{F})$, and $S(k, n) = O(n \log^{k-2+\epsilon} n)$ for some $0 < \epsilon < 1$.

11.3.8.2 Other Range Searching Problems

There are other range searching problems, called the simplex range searching problem and the half-space range searching problem that have been well studied. A simplex range in $\Re^k$ is a range whose boundary is specifed by $k + 1$ hyperplanes. In two dimensions it is a triangle.

The report-mode half-space range searching problem in the plane is optimally solved by Chazelle et al. [1985] in $Q(n) = O(\log n + \mathcal{F})$ time and $S(n) = O(n)$ space, using geometric duality transform. But this method does not generalize to higher dimensions. For $k = 3$, Chazelle and Preparata [1986] obtained an optimal $O(\log n + \mathcal{F})$ time algorithm using $O(n \log n)$ space. Agarwal and Matoušek [1995] obtained a more general result for this problem: for $n \leq m \leq n^{\lfloor k/2 \rfloor}$, with $O(m^{1+\epsilon})$ space and preprocessing,

$Q(k, n) = O((n/m^{1/\lfloor k/2 \rfloor}) \log n + \mathcal{F})$. As the half-space range searching problem is also decomposable (cf. earlier subsection on dynamization) standard dynamization techniques can be applied.

A general method for simplex range searching is to use the notion of the *partition tree*. The search space is partitioned in a hierarchical manner using cutting hyperplanes, and a search structure is built in a tree structure. Willard [1982] gave a sublinear time algorithm for count-mode half-space query in $O(n^{\alpha})$ time using linear space, where $\alpha \approx 0.774$, for $k = 2$. Using Chazelle's cutting theorem Matoušek showed that for k-dimensions there is a linear space search structure for the simplex range searching problem with query time $O(n^{1-1/k})$, which is optimal in two dimensions and within $O(\log n)$ factor of being optimal for $k > 2$. For more detailed information regarding geometric range searching see Matoušek [1994].

The preceding discussion is restricted to the case in which the database is a collection of points. One may consider other kinds of objects, such as line segments, rectangles, triangles, etc., depending on the needs of the application. The inverse of the orthogonal range searching problem is that of the *point enclosure searching problem*. Consider a collection of isothetic rectangles. The point enclosure searching problem is to find all rectangles that contain the given query point q. We can cast these problems as the *intersection searching* problems, i.e., given a set S of objects and a query object q, find a subset $\mathcal{F}$ of S such that for any $f \in \mathcal{F}$, $f \cap q \neq \emptyset$. We then have the rectangle enclosure searching problem, rectangle containment problem, segment intersection searching problem, etc. We list only a few references about these problems [Bistiolas et al. 1993, Imai and Asano 1987, Lee and Preparata 1982]. Janardan and Lopez [1993] generalized intersection searching in the following manner. The database is a collection of *groups* of objects, and the problem is to find all groups of objects intersecting a query object. A group is considered to be intersecting the query object if any object in the group intersects the query object. When each group has only one object, this reduces to the ordinary searching problems.

11.4 Conclusion

We have covered in this chapter a wide spectrum of topics in computational geometry, including several major problem solving paradigms developed to date and a variety of geometric problems. These paradigms include incremental construction, plane sweep, geometric duality, locus, divide-and-conquer, prune-and-search, dynamization, and random sampling. The topics included here, i.e., convex hull, proximity, point location, motion planning, optimization, decomposition, intersection, and searching, are not meant to be exhaustive. Some of the results presented are classic, and some of them represent the state of the art of this field. But they may also become classic in months to come. The reader is encouraged to look up the literature in major computational geometry journals and conference proceedings given in the references. We have not discussed parallel computational geometry, which has an enormous amount of research findings. Atallah [1992] gave a survey on this topic.

We hope that this treatment will provide sufficient background information about this field and that researchers in other science and engineering disciplines may find it helpful and apply some of the results to their own problem domains.

Acknowledgment

This material is based on work supported in part by the National Science Foundation under Grant CCR-9309743 and by the Office of Naval Research under Grants N00014-93-1-0272 and N00014-95-1-1007.

References

Agarwal, P., Edelsbrunner, H., Schwarzkopf, O., and Welzl, E. 1991. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete Comput. Geom.* 6(5):407–422.

Agarwal, P. and Matoušek, J. 1995. Dynamic half-space range reporting and its applications. *Algorithmica* 13(4):325–345.

Aggarwal, A., Guibas, L. J., Saxe, J., and Shor, P. W. 1989. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete Comput. Geom.* 4(6):591–604.

Alt, H. and Schwarzkopf, O. 1995. The Voronoi diagram of curved objects, pp. 89–97. In *Proc. 11th Ann. ACM Symp. Comput. Geom.*, June.

Alt, H. and Yap, C. K. 1990. Algorithmic aspect of motion planning: a tutorial, part 1 & 2. *Algorithms Rev.* 1(1, 2):43–77.

Amato, N. 1994. Determining the separation of simple polygons. *Int. J. Comput. Geom. Appl.* 4(4):457–474.

Amato, N. 1995. Finding a closest visible vertex pair between two polygons. *Algorithmica* 14(2):183–201.

Aronov, B. and Sharir, M. 1994. On translational motion planning in 3-space, pp. 21–30. In *Proc. 10th Ann. ACM Comput. Geom.*, June.

Asano, T., Asano, T., and Imai, H. 1986. Partitioning a polygonal region into trapezoids. *J. ACM* 33(2):290–312.

Asano, T., Asano, T., and Pinter, R. Y. 1986. Polygon triangulation: efficiency and minimality. *J. Algorithms* 7:221–231.

Asano, T., Guibas, L. J., and Tokuyama, T. 1994. Walking on an arrangement topologically. *Int. J. Comput. Geom. Appl.* 4(2):123–151.

Atallah, M. J. 1992. Parallel techniques for computational geometry. *Proc. of IEEE* 80(9):1435–1448.

Aurenhammer, F. 1987. Power diagrams: properties, algorithms and applications. *SIAM J. Comput.* 16(1):78–96.

Aurenhammer, F. 1990. A new duality result concerning Voronoi diagrams. *Discrete Comput. Geom.* 5(3):243–254.

Avis, D. and ElGindy, H. 1987. Triangulating point sets in space. *Discrete Comput. Geom.* 2(2):99–111.

Avis, D. and Toussaint, G. T. 1981. An efficient algorithm for decomposing a polygon into star-shaped polygons. *Pattern Recog.* 13:395–398.

Avnaim, F., Boissonnat, J. D., and Faverjon, B. 1988. A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles, pp. 67–86. In *Proc. Geom. Robotics Workshop*. J. D. Boissonnat and J. P. Laumond, eds. LNCS Vol. 391.

Balaban, I. J. 1995. An optimal algorithm for finding segments intersections, pp. 211–219. In *Proc. 11th Ann. Symp. Comput. Geom.*, June.

Bentley, J. L. 1980. Multidimensional divide-and-conquer. *Comm. ACM* 23(4):214–229.

Bentley, J. L. and Saxe, J. B. 1980. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms* 1:301–358.

Bespamyatnikh, S. N. 1995. An optimal algorithm for closest pair maintenance, pp. 152–166. In *Proc. 11th Ann. Symp. Comput. Geom.*, June.

Bieri, H. and Nef, W. 1982. A recursive plane-sweep algorithm, determining all cells of a finite division of R^d. *Computing* 28:189–198.

Bistiolas, V., Sofotassios, D., and Tsakalidis, A. 1993. Computing rectangle enclosures. *Comput. Geom. Theory Appl.* 2(6):303–308.

Boissonnat, J.-D., Sharir, M., Tagansky, B., and Yvinec, M. 1995. Voronoi diagrams in higher dimensions under certain polyhedra distance functions, pp. 79–88. In *Proc. 11th Ann. ACM Symp. Comput. Geom.*, June.

Callahan, P. and Kosaraju, S. R. 1995. Algorithms for dynamic closests pair and n-body potential fields, pp. 263–272. In *Proc. 6th ACM–SIAM Symp. Discrete Algorithms.*

Canny, J. and Reif, J. R. 1987. New lower bound techniques for robot motion planning problems, pp. 49–60. In *Proc. 28th Annual Symp. Found. Comput. Sci.*, Oct.

Chan, T. M. 1995. Output-sensitive results on convex hulls, extreme points, and related problems, pp. 10–19. In *Proc. 11th ACM Ann. Symp. Comput. Geom.*, June.

Chazelle, B. 1985. How to search in history. *Inf. Control* 64:77–99.

Chazelle, B. 1986. Filtering search: a new approach to query-answering, *SIAM J. Comput.* 15(3):703–724.

Chazelle, B. 1988. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.* 17(3):427–462.

Chazelle, B. 1991. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.* 6:485–524.
Chazelle, B. 1992. An optimal algorithm for intersecting three-dimensional convex polyhedra. *SIAM J. Comput.* 21(4):671–696.
Chazelle, B. 1993. An optimal convex hull algorithm for point sets in any fixed dimension. *Discrete Comput. Geom.* 8(2):145–158.
Chazelle, B. and Dobkin, D. P. 1987. Intersection of convex objects in two and three dimensions. *J. ACM* 34(1):1–27.
Chazelle, B. and Edelsbrunner, H. 1992. An optimal algorithm for intersecting line segments in the plane. *J. ACM* 39(1):1–54.
Chazelle, B., Edelsbrunner, H., Guibas, L. J., and Sharir, M. 1993. Diameter, width, closest line pair, and parametric searching. *Discrete Comput. Geom.* 8(2):183–196.
Chazelle, B., Edelsbrunner, H., Guibas, L. J., and Sharir, M. 1994. Algorithms for bichromatic line-segment problems and polyhedral terrains. *Algorithmica* 11(2):116–132.
Chazelle, B. and Friedman, J. 1990. A deterministic view of random sampling and its use in geometry. *Combinatorica* 10(3):229–249.
Chazelle, B. and Friedman, J. 1994. Point location among hyperplanes and unidirectional ray-shooting. *Comput. Geom. Theory Appl.* 4(2):53–62.
Chazelle, B. and Guibas, L. J. 1986. Fractional cascading: I. a data structuring technique. *Algorithmica* 1(2):133–186.
Chazelle, B., Guibas, L. J., and Lee, D. T. 1985. The power of geometric duality. *BIT* 25:76–90.
Chazelle, B. and Matoušek, J. 1993. On linear-time deterministic algorithms for optimization problems in fixed dimension, pp. 281–290. In *Proc. 4th ACM–SIAM Symp. Discrete Algorithms*.
Chazelle, B. and Preparata, F. P. 1986. Halfspace range search: an algorithmic application of k-sets. *Discrete Comput. Geom.* 1(1):83–93.
Chew, L. P. 1989. Constrained Delaunay triangulations. *Algorithmica* 4(1):97–108.
Chiang, Y.-J. and Tamassia, R. 1992. Dynamic algorithms in computational geometry. *Proc. IEEE* 80(9):1412–1434.
Clarkson, K. L. 1986. Linear programming in $O(n3^{d^2})$ time. *Inf. Proc. Lett.* 22:21–24.
Clarkson, K. L. 1988. A randomized algorithm for closest-point queries. *SIAM J. Comput.* 17(4): 830–847.
Culberson, J. C. and Reckhow, R. A. 1988. Covering polygons is hard, pp. 601–611. In *Proc. 29th Ann. IEEE Symp. Found. Comput. Sci.*
Dobkin, D. P. and Kirkpatrick, D. G. 1985. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms* 6:381–392.
Dobkin, D. P. and Munro, J. I. 1981. Optimal time minimal space selection algorithms. *J. ACM* 28(3):454–461.
Dorward, S. E. 1994. A survey of object-space hidden surface removal. *Int. J. Comput. Geom. Appl.* 4(3):325–362.
Du, D. Z. and Hwang, F. K., eds. 1992. *Computing in Euclidean Geometry*. World Scientific, Singapore.
Dyer, M. E. 1984. Linear programs for two and three variables. *SIAM J. Comput.* 13(1):31–45.
Dyer, M. E. 1986. On a multidimensional search technique and its applications to the Euclidean one-center problem. *SIAM J. Comput.* 15(3):725–738.
Edelsbrunner, H. 1985. Computing the extreme distances between two convex polygons. *J. Algorithms* 6:213–224.
Edelsbrunner, H. 1987. *Algorithms in Combinatorial Geometry*. Springer–Verlag.
Edelsbrunner, H. and Guibas, L. J. 1989. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.* 38:165–194; (1991) *Corrigendum* 42:249–251.
Edelsbrunner, H., Guibas, L. J., and Stolfi, J. 1986. Optimal point location in a monotone subdivision. *SIAM J. Comput.* 15(2):317–340.
Edelsbrunner, H., O'Rourke, J., and Seidel, R. 1986. Constructing arrangements of lines and hyperplanes with applications. *SIAM J. Comput.* 15(2):341–363.

Edelsbrunner, H. and Shi, W. 1991. An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM J. Comput.* 20(2):259–269.

Fortune, S. 1987. A sweepline algorithm for Voronoi diagrams. *Algorithmica* 2(2):153–174.

Fortune, S. 1993. Progress in computational geometry. In *Directions in Geom. Comput.*, pp. 81–128. R. Martin, ed. Information Geometers Ltd.

Ghosh, S. K. and Mount, D. M. 1991. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.* 20(5):888–910.

Guibas, L. J. and Hershberger, J. 1989. Optimal shortest path queries in a simple polygon. *J. Comput. Syst. Sci.* 39:126–152.

Guibas, L. J. and Seidel, R. 1987. Computing convolutions by reciprocal search. *Discrete Comput. Geom.* 2(2):175–193.

Handler, G. Y. and Mirchandani, P. B. 1979. *Location on Networks: Theory and Algorithm*. MIT Press, Cambridge, MA.

Hershberger, J. and Suri, S. 1993. Efficient computation of Euclidean shortest paths in the plane, pp. 508–517. In *Proc. 34th Ann. IEEE Symp. Found. Comput. Sci.*

Ho, J. M., Chang, C. H., Lee, D. T., and Wong, C. K. 1991. Minimum diameter spanning tree and related problems. *SIAM J. Comput.* 20(5):987–997.

Houle, M. E. and Toussaint, G. T. 1988. Computing the width of a set. *IEEE Trans. Pattern Anal. Machine Intelligence* PAMI-10(5):761–765.

Imai, H. and Asano, T. 1986. Efficient algorithms for geometric graph search problems. *SIAM J. Comput.* 15(2):478–494.

Imai, H. and Asano, T. 1987. Dynamic orthogonal segment intersection search. *J. Algorithms* 8(1): 1–18.

Janardan, R. and Lopez, M. 1993. Generalized intersection searching problems. *Int. J. Comput. Geom. Appl.* 3(1):39–69.

Kapoor, S. and Smid, M. 1996. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. Comput.* 25(4):775–796.

Keil, J. M. 1985. Decomposing a polygon into simpler components. *SIAM J. Comput.* 14(4):799–817.

Kirkpatrick, D. G. and Seidel, R. 1986. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299.

Klein, R. 1989. *Concrete and Abstract Voronoi Diagrams*. LNCS Vol. 400, Springer–Verlag.

Le, V. B. and Lee, D. T. 1991. Out-of-roundness problem revisited. *IEEE Trans. Pattern Anal. Machine Intelligence* 13(3):217–223.

Lee, D. T. 1978. *Proximity and Reachability in the Plan*. Ph.D. Thesis, *Tech. Rep.* R-831, Coordinated Science Lab., University of Illinois, Urbana.

Lee, D. T. and Drysdale, R. L., III 1981. Generalization of Voronoi diagrams in the plane. *SIAM J. Comput.* 10(1):73–87.

Lee, D. T. and Lin, A. K. 1986a. Computational complexity of art gallery problems. *IEEE Trans. Inf. Theory* 32(2):276–282.

Lee, D. T. and Lin, A. K. 1986b. Generalized Delaunay triangulation for planar graphs. *Discrete Comput. Geom.* 1(3):201–217.

Lee, D. T. and Preparata, F. P. 1982. An improved algorithm for the rectangle enclosure problem. *J. Algorithms* 3(3):218–224.

Lee, D. T. and Preparata, F. P. 1984. Computational geometry: a survey. *IEEE Trans. Comput.* C-33(12):1072–1101.

Lee, D. T. and Wu, V. B. 1993. Multiplicative weighted farthest neighbor Voronoi diagrams in the plane, pp. 154–168. In *Proc. Int. Workshop Discrete Math. and Algorithms*. Hong Kong, Dec.

Lee, D. T. and Wu, Y. F. 1986. Geometric complexity of some location problems. *Algorithmica* 1(2):193–211.

Lee, D. T., Yang, C. D., and Chen, T. H. 1991. Shortest rectilinear paths among weighted obstacles. *Int. J. Comput. Geom. Appl.* 1(2):109–124.

Lee, D. T., Yang, C. D., and Wong, C. K. 1996. Rectilinear paths among rectilinear obstacles. In *Perspectives in Discrete Applied Math.* K. Bogart, ed.

Lozano-Pérez, T. 1983. Spatial planning: a configuration space approach. *IEEE Trans. Comput.* C-32(2):108–120.

Mairson, H. G. and Stolfi, J. 1988. Reporting and counting intersections between two sets of line segments, pp. 307–325. In *Proc. Theor. Found. Comput. Graphics CAD*. Vol. F40, Springer–Verlag.

Matoušek, J. 1994. Geometric range searching. *ACM Computing Sur.* 26:421–461.

Matoušek, J., Sharir, M., and Welzl, E. 1992. A subexponential bound for linear programming, pp. 1–8. In *Proc. 8th Ann. ACM Symp. Comput. Geom.*

Megiddo, N. 1983a. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM* 30(4):852–865.

Megiddo, N. 1983b. Linear time algorithm for linear programming in R^3 and related problems. *SIAM J. Comput.* 12(4):759–776.

Megiddo, N. 1983c. Towards a genuinely polynomial algorithm for linear programming. *SIAM J. Comput.* 12(2):347–353.

Megiddo, N. 1984. Linear programming in linear time when the dimension is fixed. *J. ACM* 31(1):114–127.

Megiddo, N. 1986. New approaches to linear programming. *Algorithmica* 1(4):387–394.

Mehlhorn, K. 1984. *Data Structures and Algorithms*, Vol. 3, Multi-dimensional searching and computational geometry. Springer–Verlag.

Mitchell, J. S. B. 1993. Shortest paths among obstacles in the plane, pp. 308–317. In *Proc. 9th ACM Symp. Comput. Geom.*, May.

Mitchell, J. S. B., Mount, D. M., and Papadimitriou, C. H. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16(4):647–668.

Mitchell, J. S. B. and Papadimitriou, C. H. 1991. The weighted region problem: finding shortest paths through a weighted planar subdivision. *J. ACM* 38(1):18–73.

Mitchell, J. S. B., Rote, G., and Wōginger, G. 1992. Minimum link path among obstacles in the planes. *Algorithmica* 8(5/6):431–459.

Mulmuley, K. 1994. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice–Hall, Englewood Cliffs, NJ.

Nievergelt, J. and Preparata, F. P. 1982. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM* 25(10):739–747.

O'Rourke, J. 1987. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York.

O'Rourke, J. 1994. *Computational Geometry in C*. Cambridge University Press, New York.

O'Rourke, J. and Supowit, K. J. 1983. Some NP-hard polygon decomposition problems. *IEEE Trans. Inform. Theory* IT-30(2):181–190,

Overmars, M. H. 1983. *The Design of Dynamic Data Structures*. LNCS Vol. 156, Springer–Verlag.

Preparata, F. P. and Shamos, M. I. 1985. In *Computational Geometry: An Introduction*. Springer–Verlag.

Ruppert, J. and Seidel, R. 1992. On the difficulty of triangulating three-dimensional non-convex polyhedra. *Discrete Comput. Geom.* 7(3):227–253.

Schaudt, B. F. and Drysdale, R. L. 1991. Multiplicatively weighted crystal growth Voronoi diagrams, pp. 214–223. In *Proc. 7th Ann. ACM Symp. Comput. Geom.*

Schwartz, C., Smid, M., and Snoeyink, J. 1994. An optimal algorithm for the on-line closest-pair problem. *Algorithmica* 12(1):18–29.

Seo, D. Y. and Lee, D. T. 1995. On the complexity of bicriteria spanning tree problems for a set of points in the plane. *Tech. Rep. Dept.* EE/CS, Northwestern University, June.

Sharir, M. 1985. Intersection and closest-pair problems for a set of planar discs. *SIAM J. Comput.* 14(2):448–468.

Sharir, M. 1987. On shortest paths amidst convex polyhedra. *SIAM J. Comput.* 16(3):561–572.

Shermer, T. C. 1992. Recent results in art galleries. *Proc. IEEE* 80(9):1384–1399.

Sifrony, S. and Sharir, M. 1987. A new efficient motion planning algorithm for a rod in two-dimensional polygonal space. *Algorithmica* 2(4):367–402.

Smid, M. 1992. Maintaining the minimal distance of a point set in polylogarithmic time. *Discrete Comput. Geom.* 7:415–431.
Suri, S. 1990. On some link distance problems in a simple polygon. *IEEE Trans. Robotics Automation* 6(1):108–113.
Swanson, K., Lee, D. T., and Wu, V. L. 1995. An optimal algorithm for roundness determination on convex polygons. *Comput. Geom. Theory Appl.* 5(4):225–235.
Toussaint, G. T., ed. 1985. *Computational Geometry*. North–Holland.
Van der Stappen, A. F. and Overmars, M. H. 1994. Motion planning amidst fat obstacle, pp. 31–40. In *Proc. 10th Ann. ACM Comput. Geom.*, June.
van Leeuwen, J. and Wood, D. 1980. Dynamization of decomposable searching problems. *Inf. Proc. Lett.* 10:51–56.
Willard, D. E. 1982. Polygon retrieval. *SIAM J. Comput.* 11(1):149–165.
Willard, D. E. 1985. New data structures for orthogonal range queries. *SIAM J. Comput.* 14(1):232–253.
Willard, D. E. and Luecker, G. S. 1985. Adding range restriction capability to dynamic data structures. *J. ACM* 32(3):597–617.
Yao, F. F. 1994. Computational geometry. In *Handbook of Theoretical Computer Science*, Vol. A: Algorithms and Complexity, J. van Leeuwen, ed., pp. 343–389.
Yap, C. K. 1987a. An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. *Discrete Comput. Geom.* 2(4):365–393.
Yap, C. K. 1987b. Algorithmic motion planning. In *Advances in Robotics, Vol I: Algorithmic and Geometric Aspects of Robotics*. J. T. Schwartz and C. K. Yap, eds., pp. 95–143. Lawrence Erlbaum, London.

Further Information

We remark that there are new efforts being made in the applied side of algorithm development. A library of geometric software including visualization tools and applications programs is under development at the Geometry Center, University of Minnesota, and a concerted effort is being put together by researchers in Europe and in the United States to organize a system library containing primitive geometric abstract data types useful for geometric algorithm developers and practitioners.

Those who are interested in the implementations or would like to have more information about available software may consult the Proceedings of the Annual ACM Symposium on Computational Geometry, which has a video session, or the WWW page on *Geometry in Action* by David Eppstein (http://www.ics.uci.edu/˜eppstein/geom.html).

12 Randomized Algorithms

Rajeev Motwani*
Stanford University

Prabhakar Raghavan
Verity, Inc.

12.1 Introduction

A **randomized algorithm** is one that makes random choices during its execution. The behavior of such an algorithm may thus be random even on a fixed input. The design and analysis of a randomized algorithm focus on establishing that it is likely to behave well on *every* input; the likelihood in such a statement depends only on the probabilistic choices made by the algorithm during execution and not on any assumptions about the input. It is especially important to distinguish a randomized algorithm from the *average-case analysis* of algorithms, where one analyzes an algorithm assuming that its input is drawn from a fixed probability distribution. With a randomized algorithm, in contrast, no assumption is made about the input.

Two benefits of randomized algorithms have made them popular: simplicity and efficiency. For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both. In the following, we make these notions concrete through a number of illustrative examples. We assume that the reader has had undergraduate courses in algorithms and complexity, and in probability theory. A comprehensive source for randomized algorithms is the book by Motwani and Raghavan [1995]. The articles

*Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Partnership Award, an ARO MURI Grant DAAH04-96-1-0007, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

by Karp [1991], Maffioli et al. [1985], and Welsh [1983] are good surveys of randomized algorithms. The book by Mulmuley [1993] focuses on randomized geometric algorithms.

Throughout this chapter, we assume the random access memory (RAM) model of computation, in which we have a machine that can perform the following operations involving registers and main memory: input–output operations, memory–register transfers, indirect addressing, branching, and arithmetic operations. Each register or memory location may hold an integer that can be accessed as a unit, but an algorithm has no access to the representation of the number. The arithmetic instructions permitted are $+$, $-$, $\times$, and $/$. In addition, an algorithm can compare two numbers and evaluate the square root of a positive number. In this chapter, $\mathbf{E}[X]$ will denote the expectation of random variable X, and $\mathbf{Pr}[A]$ will denote the probability of event A.

12.2 Sorting and Selection by Random Sampling

Some of the earliest randomized algorithms included algorithms for sorting the set S of numbers and the related problem of finding the kth smallest element in S. The main idea behind these algorithms is the use of *random sampling*: a randomly chosen member of S is unlikely to be one of its largest or smallest elements; rather, it is likely to be near the middle. Extending this intuition suggests that a random sample of elements from S is likely to be spread roughly uniformly in S. We now describe randomized algorithms for sorting and selection based on these ideas.

Algorithm RQS

Input: A set of numbers, S.
Output: The elements of S sorted in increasing order.

1. *Choose element y uniformly at random from S: every element in S has equal probability of being chosen.*
2. *By comparing each element of S with y, determine the set S_1 of elements smaller than y and the set S_2 of elements larger than y.*
3. *Recursively sort S_1 and S_2. Output the sorted version of S_1, followed by y, and then the sorted version of S_2.*

Algorithm RQS is an example of a *randomized algorithm* — an algorithm that makes random choices during execution. It is inspired by the Quicksort algorithm due to Hoare [1962], and described in Motwani and Raghavan [1995]. We assume that the random choice in Step 1 can be made in unit time. What can we prove about the running time of RQS?

We now analyze the *expected* number of comparisons in an execution of RQS. Comparisons are performed in Step 2, in which we compare a randomly chosen element to the remaining elements. For $1 \leq i \leq n$, let $S_{(i)}$ denote the element of *rank* i (the ith smallest element) in the set S. Define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution and the value 0 otherwise. Thus, the total number of comparisons is $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$. By linearity of expectation, the expected number of comparisons is

$$\mathrm{E}\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} \mathbf{E}[X_{ij}] \tag{12.1}$$

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared during an execution. Then,

$$\mathrm{E}[X_{ij}] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij} \tag{12.2}$$

To compute p_{ij}, we view the execution of RQS as binary tree T, each node of which is labeled with a distinct element of S. The root of the tree is labeled with the element y chosen in Step 1; the left subtree of

y contains the elements in S_1 and the right subtree of y contains the elements in S_2. The structures of the two subtrees are determined recursively by the executions of RQS on S_1 and S_2. The root y is compared to the elements in the two subtrees, but no comparison is performed between an element of the left subtree and an element of the right subtree. Thus, there is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if one of these elements is an ancestor of the other.

Consider the permutation π obtained by visiting the nodes of T in increasing order of the level numbers and in a left-to-right order within each level; recall that the ith level of the tree is the set of all nodes at a distance exactly i from the root. The following two observations lead to the determination of p_{ij}:

1. There is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if $S_{(i)}$ or $S_{(j)}$ occurs earlier in the permutation π than any element $S_{(\ell)}$ such that $i < \ell < j$. To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j. If $k \notin \{i, j\}$, then $S_{(i)}$ will belong to the left subtree of $S_{(k)}$ and $S_{(j)}$ will belong to the right subtree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$. Conversely, when $k \in \{i, j\}$, there is an ancestor–descendant relationship between $S_{(i)}$ and $S_{(j)}$, implying that the two elements are compared by RQS.
2. Any of the elements $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ is equally likely to be the first of these elements to be chosen as a partitioning element and hence to appear first in π. Thus, the probability that this first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $2/(j - i + 1)$.

It follows that $p_{ij} = 2/(j - i + 1)$. By Eqs. (12.1) and (12.2), the expected number of comparisons is given by:

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j>i} p_{ij} &= \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1} \\
&\leq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k}
\end{aligned}$$

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the nth *harmonic number*, defined by $H_n = \sum_{k=1}^{n} 1/k$.

Theorem 12.1 *The expected number of comparisons in an execution of RQS is at most* $2nH_n$.

Now $H_n = \ell n\, n + \Theta(1)$, so that the expected running time of RQS is $O(n \log n)$. Note that this expected running time *holds for every input.* It is an expectation that depends only on the random choices made by the algorithm and *not* on any assumptions about the distribution of the input.

12.2.1 Randomized Selection

We now consider the use of random sampling for the problem of selecting the kth smallest element in set S of n elements drawn from a totally ordered universe. We assume that the elements of S are all distinct, although it is not very hard to modify the following analysis to allow for multisets. Let $r_S(t)$ denote the rank of element t (the kth smallest element has rank k) and recall that $S_{(i)}$ denotes the ith smallest element of S. Thus, we seek to identify $S_{(k)}$. We extend the use of this notation to subsets of S as well. The following algorithm is adapted from one due to Floyd and Rivest [1975].

Algorithm LazySelect

Input: A set, S, of n elements from a totally ordered universe and an integer, k, in $[1, n]$.
Output: The kth smallest element of S, $S_{(k)}$.

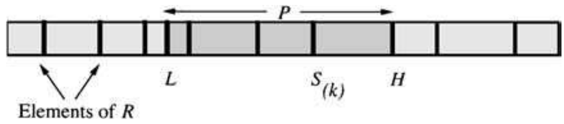


FIGURE 12.1 The LazySelect algorithm.

1. *Pick $n^{3/4}$ elements from S, chosen independently and uniformly at random with replacement; call this multiset of elements R.*
2. *Sort R in $O(n^{3/4} \log n)$ steps using any optimal sorting algorithm.*
3. *Let $x = kn^{-1/4}$. For $\ell = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ and $h = \min\{\lceil x + \sqrt{n} \rceil, n^{3/4}\}$, let $a = R_{(\ell)}$ and $b = R_{(h)}$. By comparing a and b to every element of S, determine $r_S(a)$ and $r_S(b)$.*
4. if $k < n^{1/4}$, *let $P = \{y \in S \mid y \le b\}$ and $r = k$;*
   else if $k > n - n^{1/4}$, *let $P = \{y \in S \mid y \ge a\}$ and $r = k - r_S(a) + 1$;*
   else if $k \in [n^{1/4}, n - n^{1/4}]$, *let $P = \{y \in S \mid a \le y \le b\}$ and $r = k - r_S(a) + 1$;*
   Check whether $S_{(k)} \in P$ and $|P| \le 4n^{3/4} + 2$. If not, repeat Steps 1–3 until such a set, P, is found.
5. *By sorting P in $O(|P| \log |P|)$ steps, identify P_r, which is $S_{(k)}$.*

Figure 12.1 illustrates Step 3, where small elements are at the left end of the picture and large ones are to the right. Determining (in Step 4) whether $S_{(k)} \in P$ is easy because we know the ranks $r_S(a)$ and $r_S(b)$ and we compare either or both of these to k, depending on which of the three *if* statements in Step 4 we execute. The sorting in Step 5 can be performed in $O(n^{3/4} \log n)$ steps.

Thus, the idea of the algorithm is to identify two elements a and b in S such that both of the following statements hold with high probability:

1. The element $S_{(k)}$ that we seek is in P, the set of elements between a and b.
2. The set P of elements is not very large, so that we can sort P inexpensively in Step 5.

As in the analysis of RQS, we measure the running time of LazySelect in terms of the number of comparisons performed by it. The following theorem is established using the *Chebyshev bound* from elementary probability theory; a full proof can be found in Motwani and Raghavan [1995].

Theorem 12.2 *With probability $1 - O(n^{-1/4})$, LazySelect finds $S_{(k)}$ on the first pass through Steps 1–5 and thus performs only $2n + o(n)$ comparisons.*

This adds to the significance of LazySelect — the best-known deterministic selection algorithms use $3n$ comparisons in the worst case and are quite complicated to implement.

12.3 A Simple Min-Cut Algorithm

Two events $\mathcal{E}_1$ and $\mathcal{E}_2$ are said to be *independent* if the probability that they both occur is given by

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2] = \Pr[\mathcal{E}_1] \times \Pr[\mathcal{E}_2] \tag{12.3}$$

More generally, when $\mathcal{E}_1$ and $\mathcal{E}_2$ are not necessarily independent,

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2] = \Pr[\mathcal{E}_1 \mid \mathcal{E}_2] \times \Pr[\mathcal{E}_2] = \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \times \Pr[\mathcal{E}_1] \tag{12.4}$$

where $\Pr[\mathcal{E}_1 \mid \mathcal{E}_2]$ denotes the *conditional probability* of $\mathcal{E}_1$ given $\mathcal{E}_2$. When a collection of events is not independent, the probability of their intersection is given by the following generalization of Eq. (12.4):

$$\Pr\left[\bigcap_{i=1}^{k} \mathcal{E}_i\right] = \Pr[\mathcal{E}_1] \times \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \times \Pr[\mathcal{E}_3 \mid \mathcal{E}_1 \cap \mathcal{E}_2] \cdots \Pr\left[\mathcal{E}_k \,\middle|\, \bigcap_{i=1}^{k-1} \mathcal{E}_i\right] \tag{12.5}$$

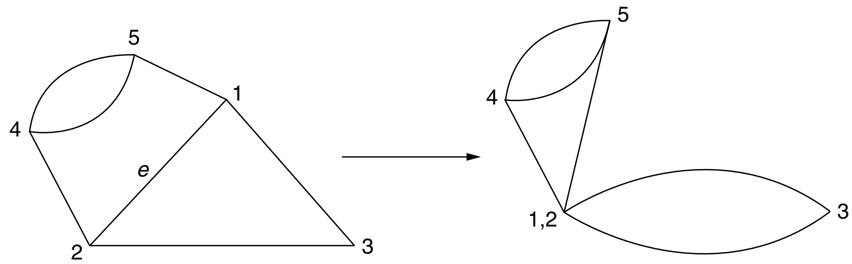


FIGURE 12.2 A step in the min-cut algorithm; the effect of contracting edge $e = (1, 2)$ is shown.

Let G be a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimum cardinality. We now study a simple algorithm due to Karger [1993] for finding a min-cut of a graph.

We repeat the following step: Pick an edge uniformly at random and merge the two vertices at its end points. If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Remove edges between vertices that are merged, so that there are never any self-loops. This process of merging the two endpoints of an edge into a single vertex is called the *contraction* of that edge. See Figure 12.2. With each contraction, the number of vertices of G decreases by one. Note that as long as at least two vertices remain, an edge contraction does not reduce the min-cut size in G. The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut. What is the probability that this algorithm finds a min-cut?

Definition 12.1 For any vertex v in the multigraph G, the *neighborhood* of G, denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v. The *degree* of v, denoted $d(v)$, is the number of edges incident on v. For the set S of vertices of G, the neighborhood of S, denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

Note that $d(v)$ is the same as the cardinality of $\Gamma(v)$ when there are no self-loops or multiple edges between v and any of its neighbors.

Let k be the min-cut size and let C be a particular min-cut with k edges. Clearly, G has at least $kn/2$ edges (otherwise there would be a vertex of degree less than k, and its incident edges would be a min-cut of size less than k). We bound from below the probability that no edge of C is ever contracted during an execution of the algorithm, so that the edges surviving until the end are exactly the edges in C.

For $1 \leq i \leq n-2$, let $\mathcal{E}_i$ denote the event of *not* picking an edge of C at the ith step. The probability that the edge randomly chosen in the first step is in C is at most $k/(nk/2) = 2/n$, so that $\Pr[\mathcal{E}_1] \geq 1 - 2/n$. Conditioned on the occurrence of $\mathcal{E}_1$, there are at least $k(n-1)/2$ edges during the second step so that $\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - 2/(n-1)$. Extending this calculation, $\Pr[\mathcal{E}_i \mid \cap_{j=1}^{i-1} \mathcal{E}_j] \geq 1 - 2/(n-i+1)$. We now invoke Eq. (12.5) to obtain

$$\Pr\left[\bigcap_{i=1}^{n-2} \mathcal{E}_i\right] \geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}$$

Our algorithm may err in declaring the cut it outputs to be a min-cut. But the probability of discovering a particular min-cut (which may in fact be the unique min-cut in G) is larger than $2/n^2$, so that the probability of error is at most $1 - 2/n^2$. Repeating the preceding algorithm $n^2/2$ times and making independent random choices each time, the probability that a min-cut is not found in any of the $n^2/2$

attempts is [by Eq. (12.3)], at most

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} < \frac{1}{e}$$

By this process of repetition, we have managed to reduce the probability of failure from $1 - 2/n^2$ to less than $1/e$. Further executions of the algorithm will make the failure probability arbitrarily small (the only consideration being that repetitions increase the running time). Note the extreme simplicity of this randomized min-cut algorithm. In contrast, most deterministic algorithms for this problem are based on network flow and are considerably more complicated.

12.3.1 Classification of Randomized Algorithms

The randomized sorting algorithm and the min-cut algorithm exemplify two different types of randomized algorithms. The sorting algorithm *always* gives the correct solution. The only variation from one run to another is its running time, whose distribution we study. Such an algorithm is called a **Las Vegas algorithm**.

In contrast, the min-cut algorithm may sometimes produce a solution that is incorrect. However, we prove that the probability of such an error is bounded. Such an algorithm is called a **Monte Carlo algorithm**. We observe a useful property of a Monte Carlo algorithm: If the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time. In some randomized algorithms, both the running time and the quality of the solution are random variables; sometimes these are also referred to as Monte Carlo algorithms. The reader is referred to Motwani and Raghavan [1995] for a detailed discussion of these issues.

12.4 Foiling an Adversary

A common paradigm in the design of randomized algorithms is that of *foiling an adversary*. Whereas an adversary might succeed in defeating a **deterministic algorithm** with a carefully constructed *bad* input, it is difficult for an adversary to defeat a randomized algorithm in this fashion. Due to the random choices made by the randomized algorithm, the adversary cannot, while constructing the input, predict the precise behavior of the algorithm. An alternative view of this process is to think of the randomized algorithm as first picking a series of random numbers, which it then uses in the course of execution as needed. In this view, we can think of the random numbers chosen at the start as *selecting* one of a family of deterministic algorithms. In other words, a randomized algorithm can be thought of as a probability distribution on deterministic algorithms. We illustrate these ideas in the setting of AND–OR *tree evaluation*; the following algorithm is due to Snir [1985].

For our purposes, an AND–OR tree is a rooted complete binary tree in which internal nodes at even distance from the root are labeled AND and internal nodes at odd distance are labeled OR. Associated with each leaf is a Boolean *value*. The *evaluation* of the game tree is the following process. Each leaf *returns* the value associated with it. Each OR node returns the Boolean OR of the values returned by its children, and each AND node returns the Boolean AND of the values returned by its children. At each step, an evaluation algorithm chooses a leaf and reads its value. We do not charge the algorithm for any other computation. We study the number of such steps taken by an algorithm for evaluating an AND–OR tree, the worst case being taken over all assignments of Boolean values of the leaves.

Let T_k denote an AND-OR tree in which every leaf is at distance $2k$ from the root. Thus, any root-to-leaf path passes through k AND nodes (including the root itself) and k OR nodes, and there are 2^{2k} leaves. An algorithm begins by specifying a leaf whose value is to be read at the first step. Thereafter, it specifies such a leaf at each step based on the values it has read on previous steps. In a deterministic algorithm, the choice of the next leaf to be read is a deterministic function of the values at the leaves read thus far. For a randomized algorithm, this choice may be randomized. It is not difficult to show that for any deterministic evaluation algorithm, there is an instance of T_k that forces the algorithm to read the values on all 2^{2k} leaves.

We now give a simple randomized algorithm and study the expected number of leaves it reads on any instance of T_k. The algorithm is motivated by the following simple observation. Consider a single AND node with two leaves. If the node were to return 0, at least one of the leaves must contain 0. A deterministic algorithm inspects the leaves in a fixed order, and an adversary can therefore always *hide* the 0 at the second of the two leaves inspected by the algorithm. Reading the leaves in a random order foils this strategy. With probability 1/2, the algorithm chooses the hidden 0 on the first step, so that its expected number of steps is 3/2, which is better than the worst case for any deterministic algorithm. Similarly, in the case of an OR node, if it were to return a 1, then a randomized order of examining the leaves will reduce the expected number of steps to 3/2. We now extend this intuition and specify the complete algorithm.

To evaluate an AND node, v, the algorithm chooses one of its children (a subtree rooted at an OR node) at random and evaluates it by recursively invoking the algorithm. If 1 is returned by the subtree, the algorithm proceeds to evaluate the other child (again by recursive application). If 0 is returned, the algorithm returns 0 for v. To evaluate an OR node, the procedure is the same with the roles of 0 and 1 interchanged. We establish by induction on k that the expected cost of evaluating any instance of T_k is at most 3^k.

The basis ($k = 0$) is trivial. Assume now that the expected cost of evaluating any instance of T_{k-1} is at most 3^{k-1}. Consider first tree T whose root is an OR node, each of whose children is the root of a copy of T_{k-1}. If the root of T were to evaluate to 1, at least one of its children returns 1. With probability 1/2, this child is chosen first, incurring (by the inductive hypothesis) an expected cost of at most 3^{k-1} in evaluating T. With probability 1/2 both subtrees are evaluated, incurring a net cost of at most $2 \times 3^{k-1}$. Thus, the expected cost of determining the value of T is

$$\leq \frac{1}{2} \times 3^{k-1} + \frac{1}{2} \times 2 \times 3^{k-1} = \frac{3}{2} \times 3^{k-1} \tag{12.6}$$

If, on the other hand, the OR were to evaluate to 0 both children must be evaluated, incurring a cost of at most $2 \times 3^{k-1}$.

Consider next the root of the tree T_k, an AND node. If it evaluates to 1, then both its subtrees rooted at OR nodes return 1. By the discussion in the previous paragraph and by linearity of expectation, the expected cost of evaluating T_k to 1 is at most $2 \times (3/2) \times 3^{k-1} = 3^k$. On the other hand, if the instance of T_k evaluates to 0, at least one of its subtrees rooted at OR nodes returns 0. With probability 1/2 it is chosen first, and so the expected cost of evaluating T_k is at most

$$2 \times 3^{k-1} + \frac{1}{2} \times \frac{3}{2} \times 3^{k-1} \leq 3^k$$

Theorem 12.3 *Given any instance of T_k, the expected number of steps for the preceding randomized algorithm is at most 3^k.*

Because $n = 4^k$, the expected running time of our randomized algorithm is $n^{\log_4 3}$, which we bound by $n^{0.793}$. Thus, the expected number of steps is smaller than the worst case for any deterministic algorithm. Note that this is a Las Vegas algorithm and always produces the correct answer.

12.5 The Minimax Principle and Lower Bounds

The randomized algorithm of the preceding section has an expected running time of $n^{0.793}$ on any uniform binary AND–OR tree with n leaves. Can we establish that *no randomized algorithm* can have a lower expected running time? We first introduce a standard technique due to Yao [1977] for proving such lower bounds. This technique applies only to algorithms that terminate in finite time on all inputs and sequences of random choices.

The crux of the technique is to relate the running times of randomized algorithms for a problem to the running times of deterministic algorithms for the problem *when faced with randomly chosen inputs*. Consider a problem where the number of distinct inputs of a fixed size is finite, as is the number of distinct

(deterministic, terminating, and always correct) algorithms for solving that problem. Let us define the **distributional complexity** of the problem at hand as the expected running time of the best deterministic algorithm for the worst distribution on the inputs. Thus, we envision an adversary choosing a probability distribution on the set of possible inputs and study the best deterministic algorithm for this distribution. Let $\boldsymbol{p}$ denote a probability distribution on the set $\mathcal{I}$ of inputs. Let the random variable $C(I_{\boldsymbol{p}}, A)$ denote the running time of deterministic algorithm $A \in \mathcal{A}$ on an input chosen according to $\boldsymbol{p}$. Viewing a randomized algorithm as a probability distribution $\boldsymbol{q}$ on the set $\mathcal{A}$ of deterministic algorithms, we let the random variable $C(I, A_{\boldsymbol{q}})$ denote the running time of this randomized algorithm on the worst-case input.

Proposition 12.1 (Yao's Minimax Principle) For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$,

$$\min_{A \in \mathcal{A}} \mathrm{E}[C(I_{\boldsymbol{p}}, A)] \leq \max_{I \in \mathcal{I}} \mathrm{E}[C(I, A_{\boldsymbol{q}})]$$

In other words, the expected running time of the optimal deterministic algorithm for an arbitrarily chosen input distribution $\boldsymbol{p}$ is a lower bound on the expected running time of the optimal (Las Vegas) randomized algorithm for Π. Thus, to prove a lower bound on the randomized complexity, it suffices to choose any distribution $\boldsymbol{p}$ on the input and prove a lower bound on the expected running time of deterministic algorithms for that distribution. The power of this technique lies in the flexibility in the choice of $\boldsymbol{p}$ and, more importantly, the reduction to a lower bound on deterministic algorithms. It is important to remember that the deterministic algorithm "knows" the chosen distribution $\boldsymbol{p}$.

The preceding discussion dealt only with lower bounds on the performance of Las Vegas algorithms. We briefly discuss Monte Carlo algorithms with error probability $\epsilon \in [0, 1/2]$. Let us define the distributional complexity with error ϵ, denoted $\min_{A \in \mathcal{A}} \mathrm{E}[C_\epsilon(I_{\boldsymbol{p}}, A)]$, to be the minimum expected running time of any deterministic algorithm that errs with probability at most ϵ under the input distribution $\boldsymbol{p}$. Similarly, we denote by $\max_{I \in \mathcal{I}} \mathrm{E}[C_\epsilon(I, A_{\boldsymbol{q}})]$ the expected running time (under the worst input) of any randomized algorithm that errs with probability at most ϵ (again, the randomized algorithm is viewed as probability distribution $\boldsymbol{q}$ on deterministic algorithms). Analogous to Proposition 12.1, we then have:

Proposition 12.2 For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$ and any $\epsilon \in [0, 1/2]$,

$$\frac{1}{2}\left(\min_{A \in \mathcal{A}} \mathrm{E}[C_{2\epsilon}(I_{\boldsymbol{p}}, A)]\right) \leq \max_{I \in \mathcal{I}} \mathrm{E}[C_\epsilon(I, A_{\boldsymbol{q}})]$$

12.5.1 Lower Bound for Game Tree Evaluation

We now apply Yao's minimax principle to the AND-OR tree evaluation problem. A randomized algorithm for AND-OR tree evaluation can be viewed as a probability distribution over deterministic algorithms, because the length of the computation as well as the number of choices at each step are both finite. We can imagine that all of these coins are tossed before the beginning of the execution.

The tree T_k is equivalent to a balanced binary tree, all of whose leaves are at distance $2k$ from the root and all of whose internal nodes compute the NOR function; a node returns the value 1 if both inputs are 0, and 0 otherwise. We proceed with the analysis of this tree of NORs of depth $2k$.

Let $p = (3 - \sqrt{5})/2$; each leaf of the tree is independently set to 1 with probability p. If each input to a NOR node is independently 1 with probability p, its output is 1 with probability

$$\left(\frac{\sqrt{5} - 1}{2}\right)^2 = \frac{3 - \sqrt{5}}{2} = p$$

Thus, the value of every node of NOR tree is 1 with probability p, and the value of a node is independent of the values of all of the other nodes on the same level. Consider a deterministic algorithm that is evaluating a tree furnished with such random inputs, and let v be a node of the tree whose value the algorithm is trying to determine. Intuitively, the algorithm should determine the value of one child of v before inspecting any leaf of the other subtree. An alternative view of this process is that the deterministic algorithm should inspect leaves visited in a depth-first search of the tree, except of course that it ceases to visit subtrees of node v when the value of v has been determined. Let us call such an algorithm a *depth-first pruning* algorithm, referring to the order of traversal and the fact that subtrees that supply no additional information are pruned away without being inspected. The following result is due to Tarsi [1983]:

Proposition 12.3 Let T be a NOR tree each of whose leaves is independently set to 1 with probability q for a fixed value $q \in [0, 1]$. Let $W(T)$ denote the minimum, over all deterministic algorithms, of the expected number of steps to evaluate T. Then, there is a depth-first pruning algorithm whose expected number of steps to evaluate T is $W(T)$.

Proposition 12.3 tells us that for the purposes of our lower bound, we can restrict our attention to depth-first pruning algorithms. Let $W(h)$ be the expected number of leaves inspected by a depth-first pruning algorithm in determining the value of a node at distance h from the leaves, when each leaf is independently set to 1 with probability $(3 - \sqrt{5})/2$. Clearly,

$$W(h) = W(h-1) + (1-p) \times W(h-1)$$

where the first term represents the work done in evaluating one of the subtrees of the node, and the second term represents the work done in evaluating the other subtree (which will be necessary if the first subtree returns the value 0, an event occurring with probability $1 - p$). Letting h be $\log_2 n$ and solving, we get $W(h) \geq n^{0.694}$.

Theorem 12.4 *The expected running time of any randomized algorithm that always evaluates an instance of T_k correctly is at least $n^{0.694}$, where $n = 2^{2k}$ is the number of leaves.*

Why is our lower bound of $n^{0.694}$ less than the upper bound of $n^{0.793}$ that follows from Theorem 12.3? The reason is that we have not chosen the best possible probability distribution for the values of the leaves. Indeed, in the NOR tree if both inputs to a node are 1, no reasonable algorithm will read leaves of both subtrees of that node. Thus, to prove the best lower bound we have to choose a distribution on the inputs that precludes the event that both inputs to a node will be 1; in other words, the values of the inputs are chosen at random but not independently. This stronger (and considerably harder) analysis can in fact be used to show that the algorithm of section 12.4 is optimal; the reader is referred to the paper of Saks and Wigderson [1986] for details.

12.6 Randomized Data Structures

Recent research into data structures has strongly emphasized the use of randomized techniques to achieve increased efficiency without sacrificing simplicity of implementation. An illustrative example is the randomized data structure for dynamic dictionaries called *skip list* that is due to Pugh [1990].

The dynamic dictionary problem is that of maintaining the set of keys X drawn from a totally ordered universe so as to provide efficient support of the following operations: find(q, X) — decide whether the query key q belongs to X and return the information associated with this key if it does indeed belong to X; insert(q, X) — insert the key q into the set X, unless it is already present in X; delete(q, X) — delete the key q from X, unless it is absent from X. The standard approach for solving this problem involves the use of a binary search tree and gives worst-case time per operation that is $O(\log n)$, where n is the size of X at the time the operation is performed. Unfortunately, achieving this time bound requires the use of

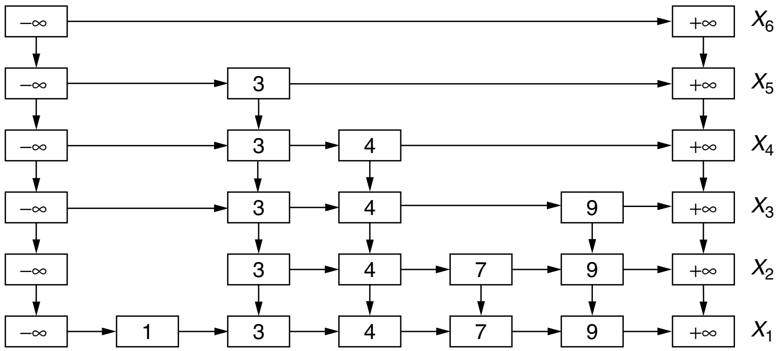


FIGURE 12.3 A skip list.

complex rebalancing strategies to ensure that the search tree remains balanced, that is, has depth $O(\log n)$. Not only does rebalancing require more effort in terms of implementation, but it also leads to significant overheads in the running time (at least in terms of the constant factors subsumed by the big-O notation). The skip list data structure is a rather pleasant alternative that overcomes both of these shortcomings.

Before getting into the details of randomized skip lists, we will develop some of the key ideas without the use of randomization. Suppose we have a totally ordered data set, $X = \{x_1 < x_2 < \cdots < x_n\}$. A *gradation of* X is a sequence of nested subsets (called *levels*)

$$X_r \subseteq X_{r-1} \subseteq \cdots \subseteq X_2 \subseteq X_1$$

such that $X_r = \emptyset$ and $X_1 = X$. Given an ordered set, X, and a gradation for it, the level of any element $x \in X$ is defined as

$$L(x) = \max\{i \mid x \in X_i\}$$

that is, $L(x)$ is the largest index i such that x belongs to the ith level of the gradation. In what follows, we will assume that two special elements $-\infty$ and $+\infty$ belong to each of the levels, where $-\infty$ is smaller than all elements in X and $+\infty$ is larger than all elements in X.

We now define an ordered list data structure with respect to a gradation of the set X. The first level, X_1, is represented as an ordered linked list, and each node x in this list has a stack of $L(x) - 1$ additional nodes directly above it. Finally, we obtain the skip list with respect to the gradation of X by introducing horizontal and vertical pointers between these nodes as illustrated in Figure 12.3. The skip list in Figure 12.3 corresponds to a gradation of the data set $X = \{1, 3, 4, 7, 9\}$ consisting of the following six levels:

$$X_6 = \emptyset$$
$$X_5 = \{3\}$$
$$X_4 = \{3, 4\}$$
$$X_3 = \{3, 4, 9\}$$
$$X_2 = \{3, 4, 7, 9\}$$
$$X_1 = \{1, 3, 4, 7, 9\}$$

Observe that starting at the ith node from the bottom in the leftmost column of nodes and traversing the horizontal pointers in order yields a set of nodes corresponding to the elements of the ith level X_i.

Additionally, we will view each level i as defining a set of *intervals*, each of which is defined as the set of elements of X spanned by a horizontal pointer at level i. The sequence of levels X_i can be viewed as successively coarser partitions of X. In Figure 12.3, the levels determine the following

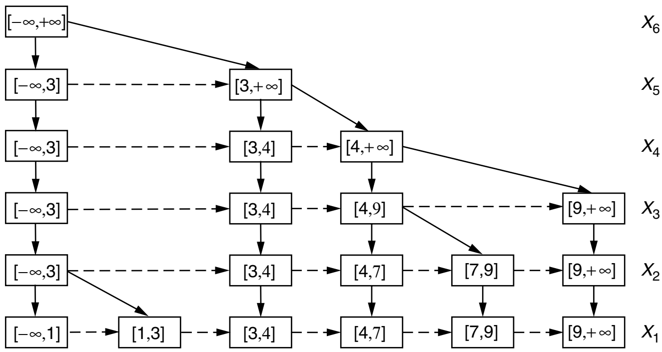


FIGURE 12.4 Tree representation of a skip list.

partitions of X into intervals:

$$\begin{aligned}
X_6 &= [-\infty, +\infty] \\
X_5 &= [-\infty, 3] \cup [3, +\infty] \\
X_4 &= [-\infty, 3] \cup [3, 4] \cup [4, +\infty] \\
X_3 &= [-\infty, 3] \cup [3, 4] \cup [4, 9] \cup [9, +\infty] \\
X_2 &= [-\infty, 3] \cup [3, 4] \cup [4, 7] \cup [7, 9] \cup [9, +\infty] \\
X_1 &= [-\infty, 1] \cup [1, 3] \cup [3, 4] \cup [4, 7] \cup [7, 9] \cup [9, +\infty]
\end{aligned}$$

An alternative view of the skip list is in terms of a tree defined by the interval partition structure, as illustrated in Figure 12.4 for the preceding example. In this tree, each node corresponds to an interval, and the intervals at a given level are represented by nodes at the corresponding level of the tree. When the interval J at level $i+1$ is a superset of the interval I at level i, then the corresponding node J has the node I as a child in this tree. Let $C(I)$ denote the number of children in the tree of a node corresponding to the interval I; that is, it is the number of intervals from the previous level that are subintervals of I. Note that the tree is not necessarily binary because the value of $C(I)$ is arbitrary. We can view the skip list as a threaded version of this tree, where each thread is a sequence of (horizontal) pointers linking together the nodes at a level into an ordered list. In Figure 12.4, the broken lines indicate the threads, and the full lines are the actual tree pointers.

Finally, we need some notation concerning the membership of element x in the intervals already defined, where x is not necessarily a member of X. For each possible x, let $I_j(x)$ be the interval at level j containing x. In the degenerate case where x lies on the boundary between two intervals, we assign it to the leftmost such interval. Observe that the nested sequence of intervals containing y,

$$I_r(y) \subseteq I_{r-1}(y) \subseteq \cdots \subseteq I_1(y),$$

corresponds to a root-leaf path in the tree corresponding to the skip list.

It remains to specify the choice of the gradation that determines the structure of a skip list. This is precisely where we introduce randomization into the structure of a skip list. The idea is to define a random gradation. Our analysis will show that, with high probability, the search tree corresponding to a random skip list is balanced, and then the dictionary operations can be efficiently implemented.

We define the *random gradation* for X as follows. Given level X_i, the next level X_{i+1} is determined by independently choosing to retain each element $x \in X_i$ with probability 1/2. The random selection process begins with $X_1 = X$ and terminates when for the first time the resulting level is empty. Alternatively, we may view the choice of the gradation as follows. For each $x \in X$, choose the level $L(x)$ independently from the geometric distribution with parameter $p = 1/2$ and place x in the levels $X_1, \ldots, X_{L(x)}$. We define r to be one more than the maximum of these level numbers. Such a random level is chosen for every element of X upon its insertion and remains fixed unit until its deletion.

We omit the proof of the following theorem bounding the space complexity of a randomized skip list. The proof is a simple exercise, and it is recommended that the reader verify this to gain some insight into the behavior of this data structure.

Theorem 12.5 *A random skip list for a set, X, of size n has expected space requirement $O(n)$.*

We will go into more detail about the time complexity of this data structure. The following lemma underlies the running time analysis.

Lemma 12.1 *The number of levels r in a random gradation of a set, X, of size n has expected value $\mathrm{E}[r] = O(\log n)$. Further, $r = O(\log n)$ with high probability.*

***Proof* 12.1** We will prove the high probability result; the bound on the expected value follows immediately from this. Recall that the level numbers $L(x)$ for $x \in X$ are independent and identically distributed (i.i.d.) random variables distributed geometrically with parameter $p = 1/2$; notationally, we will denote these random variables by $Z_1, \ldots, Z_n$. Now, the total number of levels in the skip list can be determined as

$$r = 1 + \max_{x \in X} L(x) = 1 + \max_{1 \le i \le n} Z_i$$

that is, as one more than the maximum of n i.i.d. geometric random variables.

For such geometric random variables with parameter p, it is easy to verify that for any positive real t, $\Pr[Z_i > t] \le (1 - p)^t$. It follows that

$$\Pr[\max_i Z_i > t] \le n(1-p)^t = \frac{n}{2^t}$$

because $p = 1/2$ in this case. For any $\alpha > 1$, setting $t = \alpha \log n$, we obtain

$$\Pr[r > \alpha \log n] \le \frac{1}{n^{\alpha - 1}} \qquad \Box$$

We can now infer that the tree representing the skip list has height $O(\log n)$ with high probability. To show that the overall search time in a skip list is similarly bounded, we must first specify an efficient implementation of the find operation. We present the implementation of the dictionary operations in terms of the tree representation; it is fairly easy to translate this back into the skip list representation.

To implement find (y, X), we must walk down the path

$$I_r(y) \subseteq I_{r-1}(y) \subseteq \cdots \subseteq I_1(y)$$

For this, at level j, starting at the node $I_j(y)$, we use the vertical pointer to descend to the leftmost child of the current interval; then, via the horizontal pointers, we move rightward until the node $I_j(y)$ is reached. Note that it is easily determined whether y belongs to a given interval or to an interval to its right. Further, in the skip list, the vertical pointers allow access only to the leftmost child of an interval, and therefore we must use the horizontal pointers to scan its children.

To determine the expected cost of find(y, X) operation, we must take into account both the number of levels and the number of intervals/nodes scanned at each level. Clearly, at level j, the number of nodes visited is no more than the number of children of $I_{j+1}(y)$. It follows that the cost of find can be bounded by

$$O\left(\sum_{j=1}^{r} (1 + C(I_j(y)))\right)$$

The following lemma shows that this quantity has expectation bounded by $O(\log n)$.

Lemma 12.2 *For any y, let $I_r(y), \ldots, I_1(y)$ be the search path followed by* find(y, X) *in a random skip list for a set, X, of size n. Then,*

$$E\left[\sum_{j=1}^{r}(1 + C(I_j(y)))\right] = O(\log n)$$

***Proof* 12.2** We begin by showing that for any interval I in a random skip list, $E[C(I)] = O(1)$. By Lemma 12.1, we are guaranteed that $r = O(\log n)$ with his probability, and so we will obtain the desired bound. It is important to note that we really do need the high-probability bound on Lemma 12.1 because it is incorrect to multiply the expectation of r with that of $1 + C(I)$ (the two random variables need not be independent). However, in the approach we will use, because $r > \alpha \log n$ with probability at most $1/n^{\alpha-1}$ and $\sum_j (1+C(I_j(y))) = O(n)$, it can be argued that the case $r > \alpha \log n$ does not contribute significantly to the expectation of $\sum_j C(I_j(y))$.

To show that the expected number of children of interval J at level i is bounded by a constant, we will show that the expected number of siblings of J (children of its parent) is bounded by a constant; in fact, we will bound only the number of right siblings because the argument for the number of left siblings is identical. Let the intervals to the right of J be the following:

$$J_1 = [x_1, x_2]; J_2 = [x_2, x_3]; \ldots; J_k = [x_k, +\infty]$$

Because these intervals exist at level i, each of the elements $x_1, \ldots, x_k$ belongs to X_i. If J has s right siblings, then it must be the case that $x_1, \ldots, x_s \notin X_{i+1}$, and $x_{s+1} \in X_{i+1}$. The latter event occurs with probability $1/2^{s+1}$ because each element of X_i is independently chosen to be in X_{i+1} with probability 1/2. Clearly, the number of right siblings of J can be viewed as a random variable that is geometrically distributed with parameter 1/2. It follows that the expected number of right siblings of J is at most 2. □

Consider now the implementation of the insert and delete operations. In implementing the operation insert(y, X), we assume that a random level, $L(y)$, is chosen for y as described earlier. If $L(y) > r$, then we start by creating new levels from $r + 1$ to $L(y)$ and then redefine r to be $L(y)$. This requires $O(1)$ time per level because the new levels are all empty prior to the insertion of y. Next we perform find(y, X) and determine the search path $I_r(y), \ldots, I_1(y)$, where r is updated to its new value if necessary. Given this search path, the insertion can be accomplished in time $O(L(y))$ by splitting around y the intervals $I_1(y), \ldots, I_{L(y)}(y)$ and updating the pointers as appropriate. The delete operation is the converse of the insert operation; it involves performing find(y, X) followed by collapsing the intervals that have y as an endpoint. Both operations incur costs that are the cost of a find operation and additional cost proportional to $L(y)$. By Lemmas 12.1 and 12.2, we obtain the following theorem.

Theorem 12.6 *In a random skip list for a set, X, of size n, the operations* find, insert, *and* delete *can be performed in expected time $O(\log n)$.*

12.7 Random Reordering and Linear Programming

The *linear programming problem* is a particularly notable example of the two main benefits of randomization: simplicity and speed. We now describe a simple algorithm for linear programming based on a paradigm for randomized algorithms known as *random reordering*. For many problems, it is possible to design natural algorithms based on the following idea. Suppose that the input consists of n elements. Given any subset of these n elements, there is a solution to the partial problem defined by these elements. If we start with the empty set and add the n elements of the input one at a time, maintaining a partial solution after each addition, we will obtain a solution to the entire problem when all of the elements have been added. The usual difficulty with this approach is that the running time of the algorithm depends

heavily on the order in which the input elements are added; for any fixed ordering, it is generally possible to force this algorithm to behave badly. The key idea behind random reordering is to *add the elements in a random order*. This simple device often avoids the pathological behavior that results from using a fixed order.

The linear programming problem is to find the extremum of a linear objective function of d real variables subject to set H of n constraints that are linear functions of these variables. The intersection of the n half-spaces defined by the constraints is a polyhedron in d-dimensional space (which may be empty, or possibly unbounded). We refer to this polyhedron as the *feasible region*. Without loss of generality [Schrijver 1986] we assume that the feasible region is nonempty and bounded. (Note that we are not assuming that we can *test* an arbitrary polyhedron for nonemptiness or boundedness; this is known to be equivalent to solving a linear program.) For a set of constraints, S, let $\mathcal{B}(S)$ denote the optimum of the linear program defined by S; we seek $\mathcal{B}(S)$.

Consider the following algorithm due to Seidel [1991]: Add the n constraints in random order, one at a time. After adding each constraint, determine the optimum subject to the constraints added so far. This algorithm also may be viewed in the following backwards manner, which will prove useful in the sequel.

Algorithm SLP

Input: A set of constraints H, and the dimension d.
Output: The optimum $\mathcal{B}(H)$.

0. *If there are only d constraints, output $\mathcal{B}(H) = H$.*
1. *Pick a random constraint $h \in H$;*
   Recursively find $\mathcal{B}(H\backslash\{h\})$.

2.1. If $\mathcal{B}(H\backslash\{h\})$ *does not violate h, output* $\mathcal{B}(H\backslash\{h\})$ *to be the optimum* $\mathcal{B}(H)$.
2.2. Else *project all of the constraints of* $H\backslash\{h\})$ *onto h and recursively solve this new linear programming problem of one lower dimension.*

The idea of the algorithm is simple. Either h (the constraint chosen randomly in Step 1) is redundant (in which case we execute Step 2.1), or it is not. In the latter case, we know that the vertex formed by $\mathcal{B}(H)$ must lie on the hyperplane bounding h. In this case, we project all of the constraints of $H\backslash\{h\}$ onto h and solve this new linear programming problem (which has dimension $d-1$).

The optimum $\mathcal{B}(H)$ is defined by d constraints. At the top level of recursion, the probability that random constraint h violates $\mathcal{B}(H\backslash\{h\})$ is at most d/n. Let $T(n,d)$ denote an upper bound on the expected running time of the algorithm for any problem with n constraints in d dimensions. Then, we may write

$$T(n,d) \leq T(n-1,d) + O(d) + \frac{d}{n}[O(dn) + T(n-1,d-1)] \tag{12.7}$$

In Equation (12.7), the first term on the right denotes the cost of recursively solving the linear program defined by the constraints in $H\backslash\{h\}$. The second accounts for the cost of checking whether h violates $\mathcal{B}(H\backslash\{h\})$. With probability d/n it does, and this is captured by the bracketed expression, whose first term counts the cost of projecting all of the constraints onto h. The second counts the cost of (recursively) solving the projected problem, which has one fewer constraint and dimension. The following theorem may be verified by substitution and proved by induction.

Theorem 12.7 *There is a constant b such that the recurrence (12.7) satisfies the solution $T(n,d) \leq bnd!$.*

In contrast, if the choice in Step 1 of SLP were not random, the recurrence (12.7) would be

$$T(n,d) \leq T(n-1,d) + O(d) + O(dn) + T(n-1,d-1) \tag{12.8}$$

whose solution contains a term that grows quadratically in n.

12.8 Algebraic Methods and Randomized Fingerprints

Some of the most notable randomized results in theoretical computer science, particularly in complexity theory, have involved a nontrivial combination of randomization and algebraic methods. In this section, we describe a fundamental randomization technique based on algebraic ideas. This is the randomized fingerprinting technique, originally due to Freivalds [1977], for the verification of identities involving matrices, polynomials, and integers. We also describe how this generalizes to the so-called Schwartz–Zippel technique for identities involving multivariate polynomials (independently due to Schwartz [1987] and Zippel [1979]; see also DeMillo and Lipton [1978]. Finally, following Lovász [1979], we apply the technique to the problem of detecting the existence of perfect matchings in graphs.

The *fingerprinting* technique has the following general form. Suppose we wish to decide the equality of two elements x and y drawn from some large universe U. Assuming any reasonable model of computation, this problem has a deterministic complexity $\Omega(\log|U|)$. Allowing randomization, an alternative approach is to choose a random function from U into a smaller space V such that with high probability x and y are identical if and only if their images in V are identical. These images of x and y are said to be their *fingerprints*, and the equality of fingerprints can be verified in time $O(\log|V|)$. Of course, for any fingerprint function the average number of elements of U mapped to an element of V is $|U|/|V|$; thus, it would appear impossible to find good fingerprint functions that work for arbitrary or worst-case choices of x and y. However, as we will show subsequently, when the identity checking is required to be correct only for x and y chosen from the small subspace S of U, particularly a subspace with some algebraic structure, it is possible to choose good fingerprint functions without any a priori knowledge of the subspace, provided the size of V is chosen to be comparable to the size of S.

Throughout this section, we will be working over some unspecified field $\mathcal{F}$. Because the randomization will involve uniform sampling from a finite subset of the field, we do not even need to specify whether the field is finite. The reader may find it helpful in the infinite case to assume that $\mathcal{F}$ is the field $\mathcal{Q}$ of rational numbers and in the finite case to assume that $\mathcal{F}$ is $\mathcal{Z}_p$, the field of integers modulo some prime number p.

12.8.1 Freivalds' Technique and Matrix Product Verification

We begin by describing a fingerprinting technique for verifying matrix product identities. Currently, the fastest algorithm for matrix multiplication (due to Coppersmith and Winograd [1990]) has running time $O(n^{2.376})$, improving significantly on the obvious $O(n^3)$ time algorithm; however, the fast matrix multiplication algorithm has the disadvantage of being extremely complicated. Suppose we have an implementation of the fast matrix multiplication algorithm and, given its complex nature, are unsure of its correctness. Because program verification appears to be an intractable problem, we consider the more reasonable goal of verifying the correctness of the output produced by executing the algorithm on specific inputs. (This notion of verifying programs on specific inputs is the basic tenet of the theory of *program checking* recently formulated by Blum and Kannan [1989].) More concretely, suppose we are given three $n \times n$ matrices $\boldsymbol{X}, \boldsymbol{Y}$, and $\boldsymbol{Z}$ over field $\mathcal{F}$, and would like to verify that $\boldsymbol{XY} = \boldsymbol{Z}$. Clearly, it does not make sense to use a simpler but slower matrix multiplication algorithm for the verification, as that would defeat the whole purpose of using the fast algorithm in the first place. Observe that, in fact, there is no need to recompute $\boldsymbol{Z}$; rather, we are merely required to verify that the product of $\boldsymbol{X}$ and $\boldsymbol{Y}$ is indeed equal to $\boldsymbol{Z}$. Freivalds' technique gives an elegant solution that leads to an $O(n^2)$ time randomized algorithm with bounded error probability.

The idea is to first pick the random vector $\boldsymbol{r} \in \{0, 1\}^n$, that is, each component of $\boldsymbol{r}$ is chosen independently and uniformly at random from the set $\{0, 1\}$ consisting of the additive and multiplicative identities of the field $\mathcal{F}$. Then, in $O(n^2)$ time, we can compute $\boldsymbol{y} = \boldsymbol{Yr}, \boldsymbol{x} = \boldsymbol{Xy} = \boldsymbol{XYr}$, and $\boldsymbol{z} = \boldsymbol{Zr}$. We would like to claim that the identity $\boldsymbol{XY} = \boldsymbol{Z}$ can be verified merely by checking that $\boldsymbol{x} = \boldsymbol{z}$. Quite clearly, if $\boldsymbol{XY} = \boldsymbol{Z}$, then $\boldsymbol{x} = \boldsymbol{z}$; unfortunately, the converse is not true in general. However, given the random choice of $\boldsymbol{r}$, we can show that for $\boldsymbol{XY} \neq \boldsymbol{Z}$, the probability that $\boldsymbol{x} \neq \boldsymbol{z}$ is at least 1/2. Observe that the fingerprinting algorithm errs only if $\boldsymbol{XY} \neq \boldsymbol{Z}$ but $\boldsymbol{x}$ and $\boldsymbol{z}$ turn out to be equal, and this has a bounded probability.

Theorem 12.8 *Let X, Y, and Z be $n \times n$ matrices over some field $\mathcal{F}$ such that $XY \neq Z$; further, let r be chosen uniformly at random from $\{0, 1\}^n$ and define $x = XYr$ and $z = Zr$. Then,*

$$\Pr[x = z] \leq 1/2$$

***Proof* 12.3** Define $W = XY - Z$ and observe that W is not the all-zeroes matrix. Because $Wr = XYr - Zr = x - z$, the event $x = z$ is equivalent to the event that $Wr = 0$. Assume, without loss of generality, that the first row of W has a nonzero entry and that the nonzero entries in that row precede all of the zero entries. Define the vector w as the first row of W, and assume that the first $k > 0$ entries in w are nonzero. Because the first component of Wr is $w^T r$, giving an upper bound on the probability that the inner product of w and r is zero will give an upper bound on the probability that $x = z$.

Observe that $w^T r = 0$ if and only if

$$r_1 = \frac{-\sum_{i=2}^{k} w_i r_i}{w_1} \tag{12.9}$$

Suppose that while choosing the random vector r, we choose $r_2, \ldots, r_n$ before choosing r_1. After the values for $r_2, \ldots, r_n$ have been chosen, the right-hand side of Equation (12.9) is fixed at some value $v \in \mathcal{F}$. If $v \notin \{0, 1\}$, then r_1 will never equal v; conversely, if $v \in \{0, 1\}$, then the probability that $r_1 = v$ is 1/2. Thus, the probability that $w^T r = 0$ is at most 1/2, implying the desired result. □

We have reduced the matrix multiplication verification problem to that of verifying the equality of two vectors. The reduction itself can be performed in $O(n^2)$ time and the vector equality can be checked in $O(n)$ time, giving an overall running time of $O(n^2)$ for this Monte Carlo procedure. The error probability can be reduced to $1/2^k$ via k independent iterations of the Monte Carlo algorithm. Note that there was nothing magical about choosing the components of the random vector r from $\{0, 1\}$, because any two distinct elements of $\mathcal{F}$ would have done equally well. This suggests an alternative approach toward reducing the error probability, as follows: Each component of r is chosen independently and uniformly at random from some subset $\mathcal{S}$ of the field $\mathcal{F}$; then, it is easily verified that the error probability is no more than $1/|\mathcal{S}|$.

Finally, note that Freivalds' technique can be applied to the verification of any matrix identity $A = B$. Of course, given A and B, just comparing their entries takes only $O(n^2)$ time. But there are many situations where, just as in the case of matrix product verification, computing A explicitly is either too expensive or possibly even impossible, whereas computing Ar is easy. The random fingerprint technique is an elegant solution in such settings.

12.8.2 Extension to Identities of Polynomials

The fingerprinting technique due to Freivalds is fairly general and can be applied to many different versions of the identity verification problem. We now show that it can be easily extended to identity verification for symbolic polynomials, where two polynomials $P_1(x)$ and $P_2(x)$ are deemed identical if they have identical coefficients for corresponding powers of x. Verifying integer or string equality is a special case because we can represent any string of length n as a polynomial of degree n by using the kth element in the string to determine the coefficient of the kth power of a symbolic variable.

Consider first the polynomial product verification problem: Given three polynomials $P_1(x)$, $P_2(x)$, $P_3(x) \in \mathcal{F}[x]$, we are required to verify that $P_1(x) \times P_2(x) = P_3(x)$. We will assume that $P_1(x)$ and $P_2(x)$ are of degree at most n, implying that $P_3(x)$ has degree at most $2n$. Note that degree n polynomials can be multiplied in $O(n \log n)$ time via fast Fourier transforms and that the evaluation of a polynomial can be done in $O(n)$ time.

The randomized algorithm we present for polynomial product verification is similar to the algorithm for matrix product verification. It first fixes set $\mathcal{S} \subseteq \mathcal{F}$ of size at least $2n + 1$ and chooses $r \in \mathcal{S}$ uniformly at random. Then, after evaluating $P_1(r)$, $P_2(r)$, and $P_3(r)$ in $O(n)$ time, the algorithm declares the identity $P_1(x)P_2(x) = P_3(x)$ to be correct if and only if $P_1(r)P_2(r) = P_3(r)$. The algorithm makes an error only

in the case where the polynomial identity is false but the value of the three polynomials at r indicates otherwise. We will show that the error event has a bounded probability.

Consider the degree $2n$ polynomial $Q(x) = P_1(x)P_2(x) - P_3(x)$. The polynomial $Q(x)$ is said to be *identically zero*, denoted by $Q(x) \equiv 0$, if each of its coefficients equals zero. Clearly, the polynomial identity $P_1(x)P_2(x) = P_3(x)$ holds if and only if $Q(x) \equiv 0$. We need to establish that if $Q(x) \not\equiv 0$, then with high probability $Q(r) = P_1(r)P_2(r) - P_3(r) \neq 0$. By elementary algebra we know that $Q(x)$ has at most $2n$ distinct roots. It follows that unless $Q(x) \equiv 0$, not more that $2n$ different choices of $r \in \mathcal{S}$ will cause $Q(r)$ to evaluate to 0. Therefore, the error probability is at most $2n/|\mathcal{S}|$. The probability of error can be reduced either by using independent iterations of this algorithm or by choosing a larger set $\mathcal{S}$. Of course, when $\mathcal{F}$ is an infinite field (e.g., the reals), the error probability can be made 0 by choosing r uniformly from the entire field $\mathcal{F}$; however, that requires an infinite number of random bits!

Note that we could also use a deterministic version of this algorithm where each choice of $r \in \mathcal{S}$ is tried once. But this involves $2n + 1$ different evaluations of each polynomial, and the best known algorithm for multiple evaluations needs $\Theta(n \log^2 n)$ time, which is more than the $O(n \log n)$ time requirement for actually performing a multiplication of the polynomials $P_1(x)$ and $P_2(x)$.

This verification technique is easily extended to a generic procedure for testing any polynomial identity of the form $P_1(x) = P_2(x)$ by converting it into the identity $Q(x) = P_1(x) - P_2(x) \equiv 0$. Of course, when P_1 and P_2 are explicitly provided, the identity can be deterministically verified in $O(n)$ time by comparing corresponding coefficients. Our randomized technique will take just as long to merely evaluate $P_1(x)$ and $P_2(x)$ at a random value. However, as in the case of verifying matrix identities, the randomized algorithm is quite useful in situations where the polynomials are implicitly specified, for example, when we have only a *black box* for computing the polynomials with no information about their coefficients, or when they are provided in a form where computing the actual coefficients is expensive. An example of the latter situation is provided by the following problem concerning the determinant of a symbolic matrix. In fact, the determinant problem will require a technique for the verification of polynomial identities of *multivariate* polynomials that we will discuss shortly.

Consider the $n \times n$ matrix $\boldsymbol{M}$. Recall that the determinant of the matrix M is defined as follows:

$$\det(\boldsymbol{M}) = \sum_{\pi \in \mathcal{S}_n} \operatorname{sgn}(\pi) \prod_{i=1}^{n} M_{i,\pi(i)} \tag{12.10}$$

where $\mathcal{S}_n$ is the symmetric group of permutations of order n, and $\operatorname{sgn}(\pi)$ is the sign of a permutation π. [The sign function is defined to be $\operatorname{sgn}(\pi) = (-1)^t$, where t is the number of pairwise exchanges required to convert the identity permutation into π.] Although the determinant is defined as a summation with $n!$ terms, it is easily evaluated in polynomial time provided that the matrix entries M_{ij} are explicitly specified. Consider the Vandermonde matrix $\boldsymbol{M}(x_1, \ldots, x_n)$, which is defined in terms of the indeterminates $x_1, \ldots, x_n$ such that $M_{ij} = x_i^{j-1}$, that is,

$$\boldsymbol{M} = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ & & & \cdot & \\ & & & \cdot & \\ & & & \cdot & \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix}$$

It is known that for the Vandermonde matrix, $\det(\boldsymbol{M}) = \prod_{i<j}(x_i - x_j)$. Consider the problem of verifying this identity without actually devising a formal proof. Computing the determinant of a symbolic matrix is infeasible as it requires dealing with a summation over $n!$ terms. However, we can formulate the identity verification problem as the problem of verifying that the polynomial $Q(x_1, \ldots, x_n) = \det(\boldsymbol{M}) - \prod_{i<j}(x_i - x_j)$ is identically zero. Based on our discussion of Freivalds' technique, it is natural to consider the substitution of random values for each x_i. Because the determinant can be computed in polynomial time for any

specific assignment of values to the symbolic variables $x_1, \ldots, x_n$, it is easy to evaluate the polynomial Q for random values of the variables. The only issue is that of bounding the error probability for this randomized test.

We now extend the analysis of Freivalds' technique for univariate polynomials to the multivariate case. But first, note that in a multivariate polynomial $Q(x_1, \ldots, x_n)$, the degree of a term is the sum of the exponents of the variable powers that define it, and the total degree of Q is the maximum over all terms of the degrees of the terms.

Theorem 12.9 *Let $Q(x_1, \ldots, x_n) \in \mathcal{F}[x_1, \ldots, x_n]$ be a multivariate polynomial of total degree m. Let $\mathcal{S}$ be a finite subset of the field $\mathcal{F}$, and let $r_1, \ldots, r_n$ be chosen uniformly and independently from $\mathcal{S}$. Then*

$$\Pr[Q(r_1 \ldots, r_n) = 0 \mid Q(x_1, \ldots, x_n) \not\equiv 0] \leq \frac{m}{|\mathcal{S}|}$$

***Proof* 12.4** We will proceed by induction on the number of variables n. The basis of the induction is the case $n = 1$, which reduces to verifying the theorem for a univariate polynomial $Q(x_1)$ of degree m. But we have already seen for $Q(x_1) \not\equiv 0$ the probability that $Q(r_1) = 0$ is at most $m/|\mathcal{S}|$, taking care of the basis.

We now assume that the induction hypothesis holds for multivariate polynomials with at most $n - 1$ variables, where $n > 1$. In the polynomial $Q(x_1, \ldots, x_n)$ we can factor out the variable x_1 and thereby express Q as

$$Q(x_1, \ldots, x_n) = \sum_{i=0}^{k} x_1^i P_i(x_2, \ldots, x_n)$$

where $k \leq m$ is the largest exponent of x_1 in Q. Given our choice of k, the coefficient $P_k(x_2, \ldots, x_n)$ of x_1^k cannot be identically zero. Note that the total degree of P_k is at most $m - k$. Thus, by the induction hypothesis, we conclude that the probability that $P_k(r_2, \ldots, r_n) = 0$ is at most $(m - k)/|\mathcal{S}|$.

Consider now the case where $P_k(r_2, \ldots, r_n)$ is indeed not equal to 0. We define the following univariate polynomial over x_1 by substituting the random values for the other variables in Q:

$$q(x_1) = Q(x_1, r_2, r_3, \ldots, r_n) = \sum_{i=0}^{k} x_1^i P_i(r_2, \ldots, r_n)$$

Quite clearly, the resulting polynomial $q(x_1)$ has degree k and is not identically zero (because the coefficient of x_i^k is assumed to be nonzero). As in the basis case, we conclude that the probability that $q(r_1) = Q(r_1, r_2, \ldots, r_n)$ evaluates to 0 is bounded by $k/|\mathcal{S}|$.

By the preceding arguments, we have established the following two inequalities:

$$\Pr[P_k(r_2, \ldots, r_n) = 0] \leq \frac{m-k}{|\mathcal{S}|}$$

$$\Pr[Q(r_1, r_2, \ldots, r_n) = 0 \mid P_k(r_2, \ldots, r_n) \neq 0] \leq \frac{k}{|\mathcal{S}|}$$

Using the elementary observation that for any two events $\mathcal{E}_1$ and $\mathcal{E}_2$, $\Pr[\mathcal{E}_1] \leq \Pr[\mathcal{E}_1 \mid \bar{\mathcal{E}}_2] + \Pr[\mathcal{E}_2]$, we obtain that the probability that $Q(r_1, r_2, \ldots, r_n) = 0$ is no more than the sum of the two probabilities on the right-hand side of the two obtained inequalities, which is $m/|S|$. This implies the desired results. □

This randomized verification procedure has one serious drawback: when working over large (or possibly infinite) fields, the evaluation of the polynomials could involve large intermediate values, leading to inefficient implementation. One approach to dealing with this problem in the case of integers is to perform all computations modulo some small random prime number; it can be shown that this does not have any adverse effect on the error probability.

12.8.3 Detecting Perfect Matchings in Graphs

We close by giving a surprising application of the techniques from the preceding section. Let $G(U, V, E)$ be a bipartite graph with two independent sets of vertices $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$ and edges E that have one endpoint in each of U and V. We define a matching in G as a collection of edges $M \subseteq E$ such that each vertex is an endpoint of at most one edge in M; further, a perfect matching is defined to be a matching of size n, that is, where each vertex occurs as an endpoint of exactly one edge in M. Any perfect matching M may be put into a one-to-one correspondence with the permutations in $\mathcal{S}_n$, where the matching corresponding to a permutation $\pi \in \mathcal{S}_n$ is given by the collection of edges $\{(u_i, v_{\pi(i)} \mid 1 \leq i \leq n\}$. We now relate the matchings of the graph to the determinant of a matrix obtained from the graph.

Theorem 12.10 *For any bipartite graph $G(U, V, E)$, define a corresponding $n \times n$ matrix $\boldsymbol{A}$ as follows:*

$$A_{ij} = \begin{cases} x_{ij} & (u_i, v_j) \in E \\ 0 & (u_i, v_j) \notin E \end{cases}$$

Let the multivariate polynomial $Q(x_{11}, x_{12}, \ldots, x_{nn})$ denote the determinant $\det(\boldsymbol{A})$*. Then G has a perfect matching if and only if $Q \not\equiv 0$.*

***Proof* 12.5** We can express the determinant of $\boldsymbol{A}$ as follows:

$$\det(\boldsymbol{A}) = \sum_{\pi \in S_n} \operatorname{sgn}(\pi) A_{1,\pi(1)} A_{2,\pi(2)} \ldots A_{n,\pi(n)}$$

Note that there cannot be any cancellation of the terms in the summation because each indeterminate x_{ij} occurs at most once in $\boldsymbol{A}$. Thus, the determinant is not identically zero if and only if there exists some permutation π for which the corresponding term in the summation is nonzero. Clearly, the term corresponding to a permutation π is nonzero if and only if $A_{i,\pi(i)} \neq 0$ for each $i, 1 \leq i \leq n$; this is equivalent to the presence in G of the perfect matching corresponding to π. □

The matrix of indeterminates is sometimes referred to as the *Edmonds matrix* of a bipartite graph. The preceding result can be extended to the case of nonbipartite graphs, and the corresponding matrix of indeterminates is called the Tutte matrix. Tutte [1947] first pointed out the close connection between matchings in graphs and matrix determinants; the simpler relation between bipartite matchings and matrix determinants was given by Edmonds [1967].

We can turn the preceding result into a simple randomized procedure for testing the existence of perfect matchings in a bipartite graph (due to Lovász [1979]) — using the algorithm from the preceding subsection, determine whether the determinant is identically zero. The running time of this procedure is dominated by the cost of computing a determinant, which is essentially the same as the time required to multiply two matrices. Of course, there are algorithms for *constructing* a maximum matching in a graph with m edges and n vertices in time $O(m\sqrt{n})$ (see Hopcroft and Karp [1973], Micali and Vazirani [1980], Vazirani [1994], and Feder and Motwani [1991]). Unfortunately, the time required to compute the determinant exceeds $m\sqrt{n}$ for small m, and so the benefit in using this randomized *decision* procedure appears marginal at best. However, this technique was extended by Rabin and Vazirani [1984, 1989] to obtain simple algorithms for the actual *construction* of maximum matchings; although their randomized algorithms for matchings are simple and elegant, they are still slower than the deterministic $O(m\sqrt{n})$ time algorithms known earlier. Perhaps more significantly, this randomized decision procedure proved to be an essential ingredient in devising fast *parallel* algorithms for computing maximum matchings [Karp et al. 1988, Mulmuley et al. 1987].

Defining Terms

Deterministic algorithm: An algorithm whose execution is completely determined by its input.

Distributional complexity: The expected running time of the best possible deterministic algorithm over the worst possible probability distribution of the inputs.

Las Vegas algorithm: A randomized algorithm that always produces correct results, with the only variation from one run to another being in its running time.

Monte Carlo algorithm: A randomized algorithm that may produce incorrect results but with bounded error probability.

Randomized algorithm: An algorithm that makes random choices during the course of its execution.

Randomized complexity: The expected running time of the best possible randomized algorithm over the worst input.

References

Aleliunas, R., Karp, R. M., Lipton, R. J., Lovász, L., and Rackoff, C. 1979. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proc. 20th Ann. Symp. Found. Comput. Sci.*, pp. 218–223. San Juan, Puerto Rico, Oct.

Aragon, C. R. and Seidel, R. G. 1989. Randomized search trees. In *Proc. 30th Ann. IEEE Symp. Found. Comput. Sci.*, pp. 540–545.

Ben-David, S., Borodin, A., Karp, R. M., Tardos, G., and Wigderson, A. 1994. On the power of randomization in on-line algorithms. *Algorithmica* 11(1):2–14.

Blum, M. and Kannan, S. 1989. Designing programs that check their work. In *Proc. 21st Annu. ACM Symp. Theory Comput.*, pp. 86–97. ACM.

Coppersmith, D. and Winograd, S. 1990. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.* 9:251–280.

DeMillo, R. A. and Lipton, R. J. 1978. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.* 7:193–195.

Edmonds, J. 1967. Systems of distinct representatives and linear algebra. *J. Res. Nat. Bur. Stand. 71B,* 4:241–245.

Feder, T. and Motwani, R. 1991. Clique partitions, graph compression and speeding-up algorithms. In *Proc. 25th Annu. ACM Symp. Theory Comput.*, pp. 123–133.

Floyd, R. W. and Rivest, R. L. 1975. Expected time bounds for selection. *Commun. ACM* 18:165–172.

Freivalds, R. 1977. Probabilistic machines can use less running time. In *Inf. Process. 77, Proc. IFIP Congress 77,* B. Gilchrist, Ed., pp. 839–842, North-Holland, Amsterdam, Aug.

Goemans, M. X. and Williamson, D. P. 1994. 0.878-approximation algorithms for MAX-CUT and MAX-2SAT. In *Proc. 26th Annu. ACM Symp. Theory Comput.*, pp. 422–431.

Hoare, C. A. R. 1962. Quicksort. *Comput. J.* 5:10–15.

Hopcroft, J. E. and Karp, R. M. 1973. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.* 2:225–231.

Karger, D. R. 1993. Global min-cuts in $\mathcal{RNC}$, and other ramifications of a simple min-cut algorithm. In *Proc. 4th Annu. ACM–SIAM Symp. Discrete Algorithms.*

Karger, D. R., Klein, P. N., and Tarjan, R. E. 1995. A randomized linear-time algorithm for finding minimum spanning trees. *J. ACM* 42:321–328.

Karger, D., Motwani, R., and Sudan, M. 1994. Approximate graph coloring by semidefinite programming. In *Proc. 35th Annu. IEEE Symp. Found. Comput. Sci.*, pp. 2–13.

Karp, R. M. 1991. An introduction to randomized algorithms. *Discrete Appl. Math.* 34:165–201.

Karp, R. M., Upfal, E., and Wigderson, A. 1986. Constructing a perfect matching is in random $\mathcal{NC}$. *Combinatorica* 6:35–48.

Karp, R. M., Upfal, E., and Wigderson, A. 1988. The complexity of parallel search. *J. Comput. Sys. Sci.* 36:225–253.

Lovász, L. 1979. On determinants, matchings and random algorithms. In *Fundamentals of Computing Theory.* L. Budach, Ed. Akademia-Verlag, Berlin.

Maffioli, F., Speranza, M. G., and Vercellis, C. 1985. Randomized algorithms. In *Combinatorial Optimization: Annotated Bibliographies*, M. O'Eigertaigh, J. K. Lenstra, and A. H. G. Rinooy Kan, Eds., pp. 89–105. Wiley, New York.

Micali, S. and Vazirani, V. V. 1980. An $O(\sqrt{|V|}|e|)$ algorithm for finding maximum matching in general graphs. In *Proc. 21st Annu. IEEE Symp. Found. Comput. Sci.*, pp. 17–27.

Motwani, R., Naor, J., and Raghavan, P. 1996. Randomization in approximation algorithms. In *Approximation Algorithms*, D. Hochbaum, Ed. PWS.

Motwani, R. and Raghavan, P. 1995. *Randomized Algorithms.* Cambridge University Press, New York.

Mulmuley, K. 1993. *Computational Geometry: An Introduction through Randomized Algorithms.* Prentice Hall, New York.

Mulmuley, K., Vazirani, U. V., and Vazirani, V. V. 1987. Matching is as easy as matrix inversion. *Combinatorica* 7:105–113.

Pugh, W. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33(6):668–676.

Rabin, M. O. 1980. Probabilistic algorithm for testing primality. *J. Number Theory* 12:128–138.

Rabin, M. O. 1983. Randomized Byzantine generals. In *Proc. 24th Annu. Symp. Found. Comput. Sci.*, pp. 403–409.

Rabin, M. O. and Vazirani, V. V. 1984. Maximum matchings in general graphs through randomization. *Aiken Computation Lab. Tech. Rep.* TR-15-84, Harvard University, Cambridge, MA.

Rabin, M. O. and Vazirani, V. V. 1989. Maximum matchings in general graphs through randomization. *J. Algorithms* 10:557–567.

Raghavan, P. and Snir, M. 1994. Memory versus randomization in on-line algorithms. *IBM J. Res. Dev.* 38:683–707.

Saks, M. and Wigderson, A. 1986. Probabilistic Boolean decision trees and the complexity of evaluating game trees. In *Proc. 27th Annu. IEEE Symp. Found. Comput. Sci.*, pp. 29–38. Toronto, Ontario.

Schrijver, A. 1986. *Theory of Linear and Integer Programming.* Wiley, New York.

Schwartz, J. T. 1987. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM* 27(4):701–717.

Seidel, R. G. 1991. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.* 6:423–434.

Sinclair, A. 1992. *Algorithms for Random Generation and Counting: A Markov Chain Approach, Progress in Theoretical Computer Science.* Birkhauser, Boston, MA.

Snir, M. 1985. Lower bounds on probabilistic linear decision trees. *Theor. Comput. Sci.* 38:69–82.

Solovay, R. and Strassen, V. 1977. A fast Monte-Carlo test for primality. *SIAM J. Comput.* 6(1):84–85. *See also* 1978. *SIAM J. Comput.* 7(Feb.):118.

Tarsi, M. 1983. Optimal search on some game trees. *J. ACM* 30:389–396.

Tutte, W. T. 1947. The factorization of linear graphs. *J. London Math. Soc.* 22:107–111.

Valiant, L. G. 1982. A scheme for fast parallel communication. *SIAM J. Comput.* 11:350–361.

Vazirani, V. V. 1994. A theory of alternating paths and blossoms for proving correctness of $O(\sqrt{V}E)$ graph maximum matching algorithms. *Combinatorica* 14(1):71–109.

Welsh, D. J. A. 1983. Randomised algorithms. *Discrete Appl. Math.* 5:133–145.

Yao, A. C.-C. 1977. Probabilistic computations: towards a unified measure of complexity. In *Proc. 17th Annu. Symp. Found. Comput. Sci.*, pp. 222–227.

Zippel, R. E. 1979. Probabilistic algorithms for sparse polynomials. In *Proc. EUROSAM 79,* Vol. 72, Lecture Notes in Computer Science., pp. 216–226. Marseille, France.

Further Information

In this section we give pointers to a plethora of randomized algorithms not covered in this chapter. The reader should also note that the examples in the text are but a (random!) sample of the many randomized

algorithms for each of the problems considered. These algorithms have been chosen to illustrate the main ideas behind randomized algorithms rather than to represent the state of the art for these problems. The reader interested in other algorithms for these problems is referred to Motwani and Raghavan [1995].

Randomized algorithms also find application in a number of other areas: in load balancing [Valiant 1982], approximation algorithms and combinatorial optimization [Goemans and Williamson 1994, Karger et al. 1994, Motwani et al. 1996], graph algorithms [Aleliunas et al. 1979, Karger et al. 1995], data structures [Aragon and Seidel 1989], counting and enumeration [Sinclair 1992], parallel algorithms [Karp et al. 1986, 1988], distributed algorithms [Rabin 1983], geometric algorithms [Mulmuley 1993], on-line algorithms [Ben-David et al. 1994, Raghavan and Snir 1994], and number-theoretic algorithms [Rabin 1983, Solovay and Strassen 1977]. The reader interested in these applications may consult these articles or Motwani and Raghavan [1995].

13

Pattern Matching and Text Compression Algorithms

Maxime Crochemore
University of Marne-la-Vallée and King's College London

Thierry Lecroq
University of Rouen

13.1 Processing Texts Efficiently

The present chapter describes a few standard algorithms used for processing texts. They apply, for example, to the manipulation of texts (text editors), to the storage of textual data (text compression), and to data retrieval systems. The algorithms of this chapter are interesting in different respects. First, they are basic components used in the implementations of practical software. Second, they introduce programming methods that serve as paradigms in other fields of computer science (system or software design). Third, they play an important role in theoretical computer science by providing challenging problems.

Although data is stored in various ways, text remains the main form of exchanging information. This is particularly evident in literature or linguistics where data is composed of huge corpora and dictionaries. This applies as well to computer science, where a large amount of data is stored in linear files. And this is also the case in molecular biology where biological molecules can often be approximated as sequences of

nucleotides or amino acids. Moreover, the quantity of available data in these fields tends to double every 18 months. This is the reason why algorithms should be efficient even if the speed of computers increases at a steady pace.

Pattern matching is the problem of locating a specific pattern inside raw data. The pattern is usually a collection of strings described in some formal language. Two kinds of textual patterns are presented: single strings and approximated strings. We also present two algorithms for matching patterns in images that are extensions of string-matching algorithms.

In several applications, texts need to be structured before being searched. Even if no further information is known about their syntactic structure, it is possible and indeed extremely efficient to build a data structure that supports searches. From among several existing data structures equivalent to represent indexes, we present the suffix tree, along with its construction.

The comparison of strings is implicit in the approximate pattern searching problem. Because it is sometimes required to compare just two strings (files or molecular sequences), we introduce the basic method based on longest common subsequences.

Finally, the chapter contains two classical text compression algorithms. Variants of these algorithms are implemented in practical compression software, in which they are often combined together or with other elementary methods. An example of mixing different methods is presented there.

The efficiency of algorithms is evaluated by their running times, and sometimes by the amount of memory space they require at runtime as well.

13.2 String-Matching Algorithms

String matching is the problem of finding one or, more generally, all the **occurrences** of a pattern in a text. The pattern and the text are both strings built over a finite alphabet (a finite set of symbols). Each algorithm of this section outputs all occurrences of the pattern in the text. The pattern is denoted by $x = x[0 \mathinner{..} m-1]$; its length is equal to m. The text is denoted by $y = y[0 \mathinner{..} n-1]$; its length is equal to n. The alphabet is denoted by Σ and its size is equal to σ.

String-matching algorithms of the present section work as follows: they first align the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern — this specific work is called an attempt or a scan, and after a whole match of the pattern or after a mismatch, they shift the pattern to the right. They repeat the same procedure again until the right end of the pattern goes beyond the right end of the text. This is called the scan and shift mechanism. We associate each attempt with the position j in the text, when the pattern is aligned with $y[j \mathinner{..} j+m-1]$.

The brute-force algorithm consists of checking, at all positions in the text between 0 and $n-m$, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern exactly one position to the right. This is the simplest algorithm, which is described in Figure 13.1.

The time complexity of the brute-force algorithm is $O(mn)$ in the worst case but its behavior in practice is often linear on specific data.

```
BF(x, m, y, n)
▷ Searching
for j ← 0 to n − m
    do i ← 0
       while i < m and x[i] = y[i + j]
           do i ← i + 1
       if i ≥ m
         then OUTPUT(j)
```

FIGURE 13.1 The brute-force string-matching algorithm.

13.2.1 Karp–Rabin Algorithm

Hashing provides a simple method for avoiding a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position of the text whether the pattern occurs, it seems to be more efficient to check only if the portion of the text aligned with the pattern "looks like" the pattern. To check the resemblance between these portions, a hashing function is used. To be helpful for the string-matching problem, the hashing function should have the following properties:

- Efficiently computable
- Highly discriminating for strings
- $hash(y[j+1\,..\,j+m])$ must be easily computable from $hash(y[j\,..\,j+m-1])$;
  $hash(y[j+1\,..\,j+m]) = \text{REHASH}(y[j], y[j+m], hash(y[j\,..\,j+m-1]))$

For a word w of length k, its symbols can be considered as digits, and we define $hash(w)$ by:

$$hash(w[0\,..\,k-1]) = (w[0] \times 2^{k-1} + w[1] \times 2^{k-2} + \cdots + w[k-1]) \bmod q$$

where q is a large number. Then, REHASH has a simple expression

$$\text{REHASH}(a, b, h) = ((h - a \times d) \times 2 + b) \bmod q$$

where $d = 2^{k-1}$ and q is the computer word-size (see Figure 13.2).

During the search for the pattern x, $hash(x)$ is compared with $hash(y[j-m+1\,..\,j])$ for $m-1 \le j \le n-1$. If an equality is found, it is still necessary to check the equality $x = y[j-m+1\,..\,j]$ symbol by symbol.

In the algorithms of Figures 13.2 and 13.3, all multiplications by 2 are implemented by shifts (operator $<<$). Furthermore, the computation of the modulus function is avoided by using the implicit modular

```
REHASH(a, b, h)
return ((h − a × d) << 1) + b
```

FIGURE 13.2 Function REHASH

```
KR(x, m, y, n)
▷ Preprocessing
d ← 1
for i ← 1 to m − 1
    do d ← d << 1
h_x ← 0
h_y ← 0
for i ← 0 to m − 1
    do h_x ← (h_x << 1) + x[i]
       h_y ← (h_y << 1) + y[i]
▷ Searching
if h_x = h_y and x = y[0 .. m − 1]
  then OUTPUT(0)
j ← m
while j < n
    do h_y ← REHASH(y[j − m], y[j], h_y)
       if h_x = h_y and x = y[j − m + 1 .. j]
         then OUTPUT(j − m + 1)
       j ← j + 1
```

FIGURE 13.3 The Karp–Rabin string-matching algorithm.

arithmetic given by the hardware that forgets carries in integer operations. Thus, q is chosen as the maximum value of an integer of the system.

The worst-case time complexity of the Karp–Rabin algorithm (Figure 13.3) is quadratic (as it is for the brute-force algorithm), but its expected running time is $O(m+n)$.

Example 13.1

Let $x =$ **ing**. Then, $hash(x) = 105 \times 2^2 + 110 \times 2 + 103 = 743$ (symbols are assimilated with their ASCII codes):

y	=	**s**	**t**	**r**	**i**	**n**	**g**		**m**	**a**	**t**	**c**	**h**	**i**	**n**	**g**
hash	=			806	797	776	743	678	585	443	746	719	766	709	736	743

13.2.2 Knuth–Morris–Pratt Algorithm

This section presents the first discovered linear-time string-matching algorithm. Its design follows a tight analysis of the brute-force algorithm, and especially the way this latter algorithm wastes the information gathered during the scan of the text.

Let us look more closely at the brute-force algorithm. It is possible to improve the length of shifts and simultaneously remember some portions of the text that match the pattern. This saves comparisons between characters of the text and of the pattern, and consequently increases the speed of the search.

Consider an attempt at position j, that is, when the pattern $x[0\,..\,m-1]$ is aligned with the segment $y[j\,..\,j+m-1]$ of the text. Assume that the first mismatch (during a left-to-right scan) occurs between symbols $x[i]$ and $y[i+j]$ for $0 \le i < m$. Then, $x[0\,..\,i-1] = y[j\,..\,i+j-1] = u$ and $a = x[i] \neq y[i+j] = b$. When shifting, it is reasonable to expect that a **prefix** v of the pattern matches some **suffix** of the portion u of the text. Moreover, if we want to avoid another immediate mismatch, the letter following the prefix v in the pattern must be different from a. (Indeed, it should be expected that v matches a suffix of ub, but elaborating along this idea goes beyond the scope of the chapter.) The longest such prefix v is called the **border** of u (it occurs at both ends of u). This introduces the notation: let $next[i]$ be the length of the longest (proper) border of $x[0\,..\,i-1]$, followed by a character c different from $x[i]$. Then, after a shift, the comparisons can resume between characters $x[next[i]]$ and $y[i+j]$ without missing any occurrence of x in y and having to backtrack on the text (see Figure 13.4).

Example 13.2

Here,

$y =$	.	.	.	**a**	**b**	**a**	**b**	**a**	**a**	**b**	.	.	.	.	.
$x =$				$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	**a**					
$x =$									$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	**a**	**b**	**a**	**b**	**a**

Compared symbols are underlined. Note that the empty string is the suitable border of **ababa**. Other borders of **ababa** are **aba** and **a**.

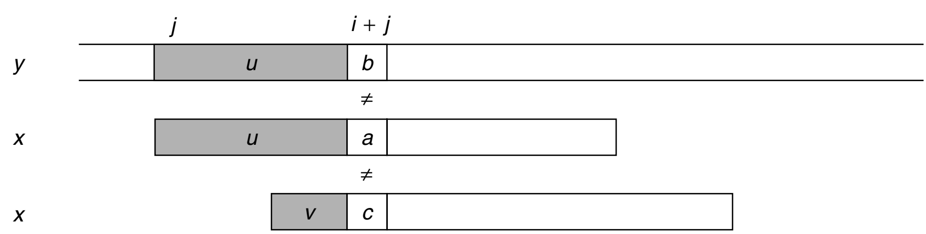


FIGURE 13.4 Shift in the Knuth–Morris–Pratt algorithm (v suffix of u).

```
KMP(x, m, y, n)
▷ Preprocessing
next ← PREKMP(x, m)
▷ Searching
i ← 0
j ← 0
while j < n
    do while i > −1 and x[i] ≠ y[j]
           do i ← next[i]
       i ← i + 1
       j ← j + 1
       if i ≥ m
         then OUTPUT(j − i)
              i ← next[i]
```

FIGURE 13.5 The Knuth–Morris–Pratt string-matching algorithm.

```
PREKMP(x, m)
i ← −1
j ← 0
next[0] ← −1
while j < m
    do while i > −1 and x[i] ≠ x[j]
           do i ← next[i]
       i ← i + 1
       j ← j + 1
       if x[i] = x[j]
         then next[j] ← next[i]
         else next[j] ← i
return next
```

FIGURE 13.6 Preprocessing phase of the Knuth–Morris–Pratt algorithm: computing *next*.

The Knuth–Morris–Pratt algorithm is displayed in Figure 13.5. The table *next* it uses is computed in $O(m)$ time before the search phase, applying the same searching algorithm to the pattern itself, as if $y = x$ (see Figure 13.6). The worst-case running time of the algorithm is $O(m + n)$ and it requires $O(m)$ extra space. These quantities are independent of the size of the underlying alphabet.

13.2.3 Boyer–Moore Algorithm

The Boyer–Moore algorithm is considered the most efficient string-matching algorithm in usual applications. A simplified version of it, or the entire algorithm, is often implemented in text editors for the search and substitute commands.

The algorithm scans the characters of the pattern from right to left, beginning with the rightmost symbol. In case of a mismatch (or a complete match of the whole pattern), it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the *bad-character shift* and the *good-suffix shift*. They are based on the following observations.

Assume that a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i+j] = b$ of the text during an attempt at position j. Then, $x[i+1\,..\,m-1] = y[i+j+1\,..\,j+m-1] = u$ and $x[i] \neq y[i+j]$. The good-suffix shift consists in aligning the **segment** $y[i+j+1\,..\,j+m-1]$ with its rightmost occurrence in x that is preceded by a character different from $x[i]$ (see Figure 13.7). If there exists no such segment, the shift consists in aligning the longest suffix v of $y[i+j+1\,..\,j+m-1]$ with a matching prefix of x (see Figure 13.8).

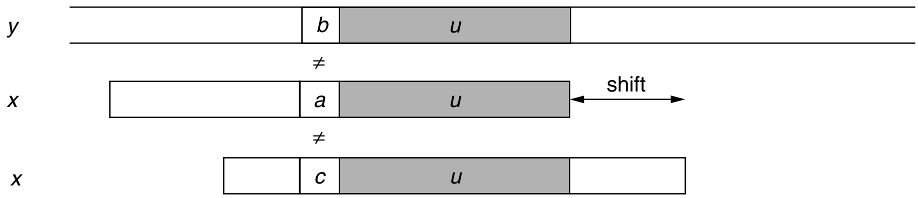


FIGURE 13.7 The good-suffix shift, when u reappears, preceded by a character different from a.

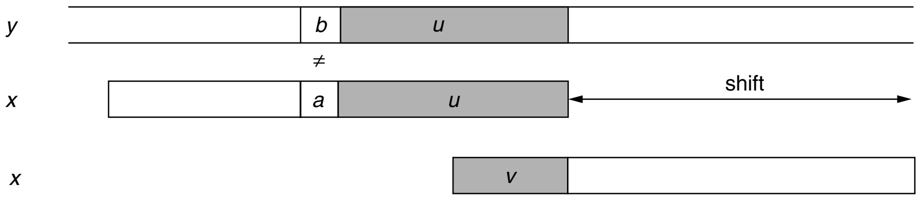


FIGURE 13.8 The good-suffix shift, when the situation of Figure 13.7 does not happen, only a suffix of u reappears as a prefix of x.

Example 13.3

Here,

$$
\begin{array}{lccccccccccccccccc}
y = & . & . & . & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & . & . & . \\
x = & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{a} & \mathbf{b} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & & & & & & \\
x = & & & & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & & &
\end{array}
$$

The shift is driven by the suffix **abba** of x found in the text. After the shift, the segment **abba** in the middle of y matches a segment of x as in Figure 13.7. The same mismatch does not recur.

Example 13.4

Here,

$$
\begin{array}{lccccccccccccccccccc}
y = & . & . & . & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & . & . \\
x = & & & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{b} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & & & & & & & & \\
x = & & & & & & & & & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & &
\end{array}
$$

The segment **abba** found in y partially matches a prefix of x after the shift, as in Figure 13.8.

The bad-character shift consists in aligning the text character $y[i + j]$ with its rightmost occurrence in $x[0 \mathinner{..} m - 2]$ (see Figure 13.9). If $y[i + j]$ does not appear in the pattern x, no occurrence of x in y can overlap the symbol $y[i + j]$, then the left end of the pattern is aligned with the character at position $i + j + 1$ (see Figure 13.10).

Example 13.5

Here,

$$
\begin{array}{lcccccccccccccc}
y = & . & . & . & . & . & . & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & . & . & . & . \\
x = & \mathbf{c} & \mathbf{d} & \mathbf{a} & \mathbf{h} & \mathbf{g} & \mathbf{f} & \underline{\mathbf{e}} & \underline{\mathbf{b}} & \underline{\mathbf{c}} & \underline{\mathbf{d}} & & & & \\
x = & & & & & \mathbf{c} & \mathbf{d} & \mathbf{a} & \mathbf{h} & \mathbf{g} & \mathbf{f} & \mathbf{e} & \mathbf{b} & \mathbf{c} & \underline{\mathbf{d}}
\end{array}
$$

The shift aligns the symbol **a** in x with the mismatch symbol **a** in the text y (Figure 13.9).

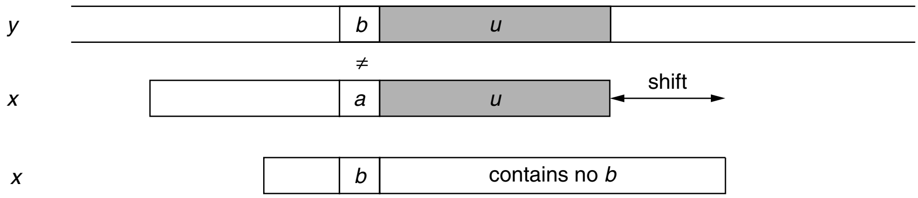


FIGURE 13.9 The bad-character shift, b appears in x.

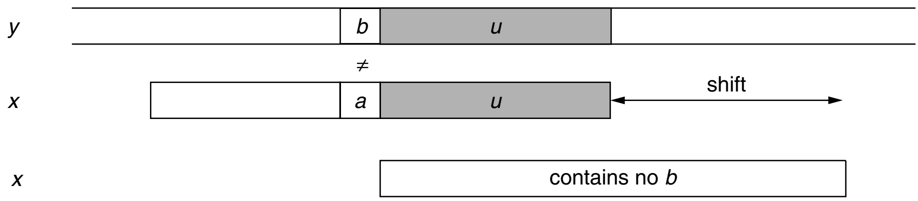


FIGURE 13.10 The bad-character shift, b does not appear in x (except possibly at $m - 1$).

```
BM(x, m, y, n)
  ▷ Preprocessing
  gs ← PREGS(x, m)
  bc ← PREBC(x, m)
  ▷ Preprocessing
  j ← 0
  while j ≤ n − m
      do i ← m − 1
         while i ≥ 0 and x[i] = y[i + j]
             do i ← i − 1
         if i < 0
           then OUTPUT(j)
         j ← max{gs[i + 1], bc[y[i + j] − m + i + 1]}
```

FIGURE 13.11 The Boyer–Moore string-matching algorithm.

Example 13.6

Here,

$y =$	.	.	.	.	.	a	b	c	d	.	.	.	.	.	.	.
$x =$	c	d	h	g	f	$\underline{\text{e}}$	$\underline{\text{b}}$	$\underline{\text{c}}$	$\underline{\text{d}}$							
$x =$							c	d	h	g	f	e	b	c	$\underline{\text{d}}$	

The shift positions the left end of x right after the symbol **a** of y (Figure 13.10).

The Boyer–Moore algorithm is shown in Figure 13.11. For shifting the pattern, it applies the maximum between the bad-character shift and the good-suffix shift. More formally, the two shift functions are defined as follows. The bad-character shift is stored in a table bc of size σ and the good-suffix shift is stored in a table gs of size $m + 1$. For $a \in \Sigma$

$$bc[a] = \begin{cases} \min\{i \mid 1 \leq i < m \text{ and } x[m-1-i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

```
PreBC(x, m)
for a ← firstLetter to lastLetter
    do bc[a] ← m
for i ← 0 to m − 2
    do bc[x[i]] ← m − 1 − i
return bc
```

FIGURE 13.12 Computation of the bad-character shift.

```
Suffixes(x, m)
suff[m − 1] ← m
g ← m − 1
for i ← m − 2 downto 0
    do if i > g and suff[i + m − 1 − f] ≠ i − g
          then suff[i] ← min{suff[i + m − 1 − f], i − g}
          else  if i < g
                  then g ← i
                f ← i
                while g ≥ 0 and x[g] = x[g + m − 1 − f]
                    do g ← g − 1
                suff[i] ← f − g
return suff
```

FIGURE 13.13 Computation of the table *suff*.

Let us define two conditions,

$$\begin{cases} cond_1(i,s): \text{ for each } k \text{ such that } i < k < m, s \geq k \text{ or } x[k-s] = x[k], \\ cond_2(i,s): \text{ if} s < i \text{ then } x[i-s] \neq x[i]. \end{cases}$$

Then, for $0 \leq i < m$,

$$gs[i+1] = \min\{s > 0 \mid cond_1(i,s) \text{ and } cond_2(i,s) \text{ hold}\}$$

and we define $gs[0]$ as the length of the smallest period of x.

To compute the table *gs*, a table *suff* is used. This table can be defined as follows: for $i = 0, 1, \ldots, m-1$,

$$suff[i] = \text{longest common suffix between } x[0\,..\,i] \text{ and } x\,.$$

It is computed in linear time and space by the function Suffixes (see Figure 13.13).

Tables *bc* and *gs* can be precomputed in time $O(m + \sigma)$ before the search phase and require an extra space in $O(m + \sigma)$ (see Figure 13.12 and Figure 13.14). The worst-case running time of the algorithm is quadratic. However, on large alphabets (relative to the length of the pattern), the algorithm is extremely fast. Slight modifications of the strategy yield linear-time algorithms (see the bibliographic notes). When searching for a^m in $(a^{m-1}b)^{\lfloor n/m \rfloor}$, the algorithm makes only $O(n/m)$ comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

13.2.4 Quick Search Algorithm

The bad-character shift used in the Boyer–Moore algorithm is not very efficient for small alphabets; but when the alphabet is large compared with the length of the pattern, as is often the case with the ASCII table and ordinary searches made under a text editor, it becomes very useful. Using it alone produces a practically very efficient algorithm that is described now.

After an attempt where x is aligned with $y[j\,..\,j+m-1]$, the length of the shift is at least equal to one. Thus, the character $y[j+m]$ is necessarily involved in the next attempt, and thus can be used for

```
PREGS(x, m)
gs ← SUFFIXES(x, m)
for i ← 0 to m − 1
    do gs[i] ← m
j ← 0
for i ← m − 1 downto −1
    do if i = −1 or suff[i] = i + 1
         then while j < m − 1 − i
                   do if gs[j] = m
                        then gs[j] ← m − 1 − i
                      j ← j + 1
for i ← 0 to m − 2
    do gs[m − 1 − suff[i]] ← m − 1 − i
return gs
```

FIGURE 13.14 Computation of the good-suffix shift.

```
QS(x, m, y, n)
▷ Preprocessing
for a ← firstLetter to lastLetter
    do bc[a] ← m + 1
for i ← 0 to m − 1
    do bc[x[i]] ← m − i
▷ Searching
j ← 0
while j ≤ n − m
    do i ← 0
       while i ≥ 0 and x[i] = y[i + j]
           do i ← i + 1
       if i ≥ m
         then OUTPUT(j)
       j ← bc[y[j + m]]
```

FIGURE 13.15 The Quick Search string-matching algorithm.

the bad-character shift of the current attempt. In the present algorithm, the bad-character shift is slightly modified to take into account the observation as follows ($a \in \Sigma$):

$$bc[a] = 1 + \begin{cases} \min\{i \mid 0 \le i < m \text{ and } x[m-1-i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

Indeed, the comparisons between text and pattern characters during each attempt can be done in any order. The algorithm of Figure 13.15 performs the comparisons from left to right. It is called Quick Search after its inventor and has a quadratic worst-case time complexity but good practical behavior.

Example 13.7

Here,

$y =$	s	t	r	i	n	g	-	m	a	t	c	h	i	n	g
$x =$	$\underline{\text{i}}$	n	g												
$x =$				$\underline{\text{i}}$	$\underline{\text{n}}$	$\underline{\text{g}}$									
$x =$								$\underline{\text{i}}$	n	g					
$x =$												$\underline{\text{i}}$	n	g	
$x =$													$\underline{\text{i}}$	$\underline{\text{n}}$	$\underline{\text{g}}$

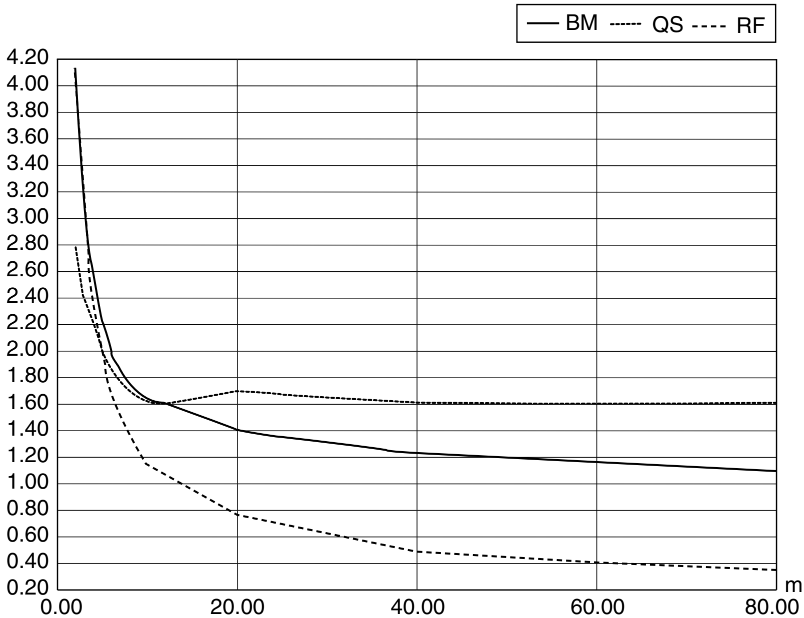


FIGURE 13.16 Running times for a DNA sequence.

The Quick Search algorithm makes only nine comparisons to find the two occurrences of **ing** inside the text of length 15.

13.2.5 Experimental Results

In Figure 13.16 and Figure 13.17, we present the running times of three string-matching algorithms: the Boyer–Moore algorithm (BM), the Quick Search algorithm (QS), and the Reverse-Factor algorithm (RF). The Reverse–Factor algorithm can be viewed as a variation of the Boyer–Moore algorithm where factors (segments) rather than suffixes of the pattern are recognized. The RF algorithm uses a data structure to store all the factors of the reversed pattern: a suffix automaton or a **suffix tree**.

Tests have been performed on various types of texts. In Figure 13.16 we show the results when the text is a DNA sequence on the four-letter alphabet of nucleotides **A**, **C**, **G**, **T**. In Figure 13.17 English text is considered.

For each pattern length, we ran a large number of searches with random patterns. The average time according to the length is shown in the two figures. The running times of both preprocessing and searching phases are added. The three algorithms are implemented in a homogeneous way in order to keep the comparison significant.

For the genome, as expected, the QS algorithm is the best for short patterns. But for long patterns it is less efficient than the BM algorithm. In this latter case, the RF algorithm achieves the best results. For rather large alphabets, as is the case for an English text, the QS algorithm remains better than the BM algorithm whatever the pattern length is. In this case, the three algorithms have similar behaviors; however, the QS is better for short patterns (which is typical of search under a text editor) and the RF is better for large patterns.

13.2.6 Aho–Corasick Algorithm

The Unix operating system provides standard text (or file) facilities. Among them is the series of **grep** commands that locate patterns in files. We describe in this section the algorithm underlying the **fgrep**

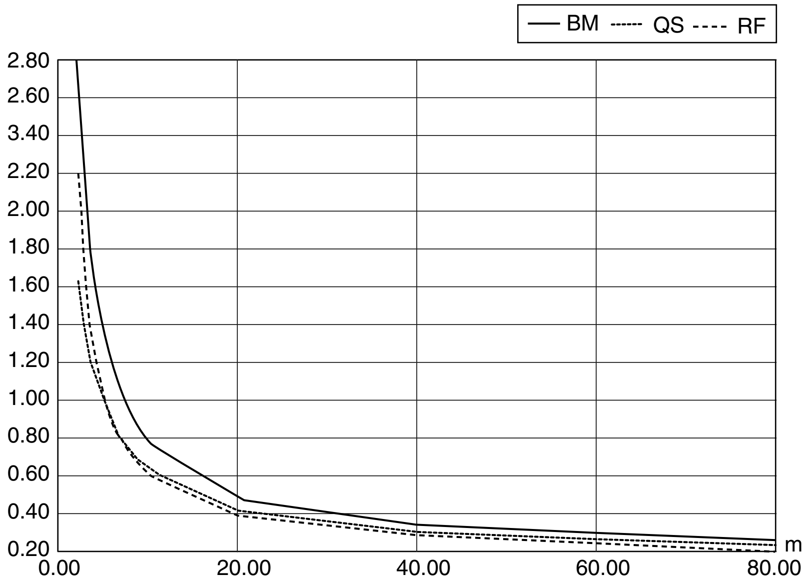


FIGURE 13.17 Running times for an English text.

```
PREAC(X, k)
Create a new node root
▷ creates a loop on the root of the trie
for a ∈ Σ
    do child(root, a) ← root
▷ enters each pattern in the trie
for i ← 0 to k − 1
    do ENTER(X[i], root)
▷ completes the trie with failure links
COMPLETE(root)
return root
```

FIGURE 13.18 Preprocessing phase of the Aho–Corasick algorithm.

command of Unix. It searches files for a finite set of strings, and can, for instance, output lines containing at least one of the strings.

If we are interested in searching for all occurrences of all patterns taken from a finite set of patterns, a first solution consists in repeating some string-matching algorithm for each pattern. If the set contains k patterns, this search runs in time $O(kn)$. The solution described in the present section and designed by Aho and Corasick runs in time $O(n \log \sigma)$. The algorithm is a direct extension of the Knuth–Morris–Pratt algorithm, and the running time is independent of the number of patterns.

Let $X = \{x_0, x_1, \ldots, x_{k-1}\}$ be the set of patterns, and let $|X| = |x_0| + |x_1| + \cdots + |x_{k-1}|$ be the total size of the set X. The Aho–Corasick algorithm first builds a **trie** $T(X)$, a digital tree recognizing the patterns of X. The trie $T(X)$ is a tree in which edges are labeled by letters and in which branches spell the patterns of X. We identify a node p in the trie $T(X)$ with the unique word w spelled by the path of $T(X)$ from its root to p. The root itself is identified with the empty word ε. Notice that if w is a node in $T(X)$ then w is a prefix of some $x_i \in X$. If w is a node in $T(X)$ and $a \in \Sigma$ then *child*(w, a) is equal to wa if wa is a node in $T(X)$; it is equal to UNDEFINED otherwise.

The function PREAC in Figure 13.18 returns the trie of all patterns. During the second phase, where patterns are entered in the trie, the algorithm initializes an output function out. It associates the singleton

```
ENTER(x, root)
r ← root
i ← 0
▷ follows the existing edges
while i < |x| and child(r, x[i]) ≠ UNDEFINED and child(r, x[i]) ≠ root
    do r ← child(r, x[i])
       i ← i + 1
▷ creates new edges
while i < |x|
    do Create a new node s
       child(r, x[i]) ← s
       r ← s
       i ← i + 1
out(r) ← {x}
```

FIGURE 13.19 Construction of the trie.

```
COMPLETE(root)
q ← empty queue
ℓ ← list of the edges (root, a, p) for any character a ∈ Σ and any node p ≠ root
while the list ℓ is not empty
    do (r, a, p) ← FIRST(ℓ)
       ℓ ← NEXT(ℓ)
       ENQUEUE(q, p)
       fail(p) ← root
while the queue q is not empty
    do r ← DEQUEUE(q)
       ℓ ← list of the edges (r, a, p) for any character a ∈ Σ and any node p
       while the list ℓ is not empty
           do (r, a, p) ← FIRST(ℓ)
              ℓ ← NEXT(ℓ)
              ENQUEUE(q, p)
              s ← fail(r)
              while child(s, a) = UNDEFINED
                  do s ← fail(s)
              fail(p) ← child(s, a)
              out(p) ← out(p) ∪ out(child(s, a))
```

FIGURE 13.20 Completion of the output function and construction of failure links.

$\{x_i\}$ with the nodes x_i ($0 \leq i < k$), and associates the empty set with all other nodes of $T(X)$ (see Figure 13.19).

Finally, the last phase of function PREAC (Figure 13.18) consists in building the failure link of each node of the trie, and simultaneously completing the output function. This is done by the function COMPLETE in Figure 13.20. The failure function *fail* is defined on nodes as follows (w is a node):

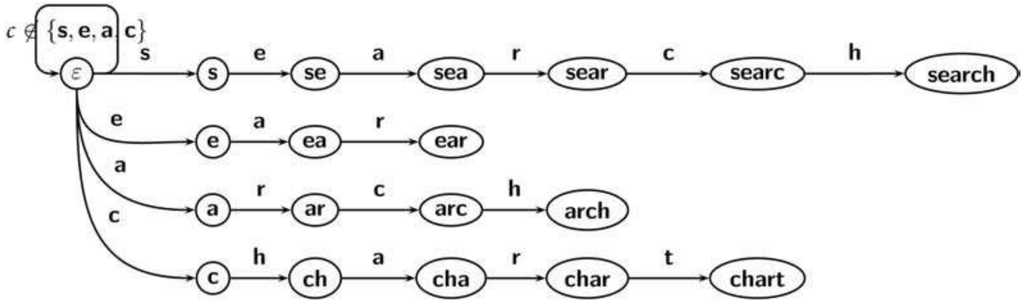
$$fail(w) = u \quad \text{where } u \text{ is the longest proper suffix of } w \text{ that belongs to } T(X).$$

Computation of failure links is done during a breadth-first traversal of $T(X)$. Completion of the output function is done while computing the failure function *fail* using the following rule:

$$\text{if } fail(w) = u \text{ then } out(w) = out(w) \cup out(u).$$

Example 13.8

Here, $X = \{$**search**, **ear**, **arch**, **chart**$\}$



nodes	ε	s	se	sea	sear	searc	search	e	ea	ear
fail	ε	ε	e	ea	ear	arc	arch	ε	a	ar

nodes	a	ar	arc	arch	c	ch	cha	char	chart	
fail	ε	ε	c	ch	ε	ε	a	ar	ε	

nodes	sear	search	ear	arch	chart
out	ear	{search, arch}	ear	arch	chart

To stop going back with failure links during the computation of the failure links, and also to overpass text characters for which no transition is defined from the root, a loop is added on the root of the trie for these symbols. This is done at the first phase of function PREAC.

After the preprocessing phase is completed, the searching phase consists in parsing all the characters of the text y with $T(X)$. This starts at the root of $T(X)$ and uses failure links whenever a character in y does not match any label of outgoing edges of the current node. Each time a node with a nonempty output is encountered, this means that the patterns of the output have been discovered in the text, ending at the current position. Then, the position is output.

An implementation of the Aho–Corasick algorithm from the previous discussion is shown in Figure 13.21. Note that the algorithm processes the text in an on-line way, so that the buffer on the text can be limited to only one symbol. Also note that the instruction $r \leftarrow \mathit{fail}(r)$ in Figure 13.21 is the exact analogue of instruction $i \leftarrow \mathit{next}[i]$ in Figure 13.5. A unified view of both algorithms exists but is beyond the scope of the chapter.

The entire algorithm runs in time $O(|X| + n)$ if the *child* function is implemented to run in constant time. This is the case for any fixed alphabet. Otherwise, a $\log\sigma$ multiplicative factor comes from access to the children nodes.

```
AC(X, k, y, n)
  ▷ Preprocessing
  r ← PREAC(X, k)
  ▷ Searching
  for j ← 0 to n − 1
      do while child(r, y[j]) = UNDEFINED
              do r ← fail(r)
         r ← child(r, y[j])
         if out(r) ≠ ∅
            then OUTPUT((out(r), j))
```

FIGURE 13.21 The complete Aho–Corasick algorithm.

13.3 Two-Dimensional Pattern Matching Algorithms

In this section we consider only two-dimensional arrays. Arrays can be thought of as bit map representations of images, where each cell of arrays contains the codeword of a pixel. The string-matching problem finds an equivalent formulation in two dimensions (and even in any number of dimensions), and algorithms of Section 13.2 can be extended to operate on arrays.

The problem now is to locate all occurrences of a two-dimensional pattern $X = X[0\,..\,m_1-1, 0\,..\,m_2-1]$ of size $m_1 \times m_2$ inside a two-dimensional text $Y = Y[0\,..\,n_1-1, 0\,..\,n_2-1]$ of size $n_1 \times n_2$. The brute-force algorithm for this problem is given in Figure 13.22. It consists in checking at all positions of $Y[0\,..\,n_1-m_1, 0\,..\,n_2-m_2]$ if the pattern occurs. This algorithm has a quadratic (with respect to the size of the problem) worst-case time complexity in $O(m_1m_2n_1n_2)$. We present in the next sections two more efficient algorithms. The first one is an extension of the Karp–Rabin algorithm (previous section). The second one solves the problem in linear time on a fixed alphabet; it uses both the Aho–Corasick and the Knuth–Morris–Pratt algorithms.

13.3.1 Zhu–Takaoka Algorithm

As for one-dimensional string matching, it is possible to check if the pattern occurs in the text only if the *aligned* portion of the text looks like the pattern. To do that, the idea is to use vertically the hash function method proposed by Karp and Rabin. To initialize the process, the two-dimensional arrays X and Y are translated into one-dimensional arrays of numbers x and y. The translation from X to x is done as follows $(0 \le i < m_2)$:

$$x[i] = hash(X[0,i]X[1,i]\cdots X[m_1-1,i])$$

and the translation from Y to y is done by $(0 \le i < m_2)$:

$$y[i] = hash(Y[0,i]Y[1,i]\cdots Y[m_1-1,i]).$$

The fingerprint y helps to find occurrences of X starting at row $j = 0$ in Y. It is then updated for each new row in the following way $(0 \le i < m_2)$:

$$\begin{aligned} &hash(Y[j+1,i]Y[j+2,i]\cdots Y[j+m_1,i]) \\ &\quad = \text{REHASH}(Y[j,i], Y[j+m_1,i], hash(Y[j,i]Y[j+1,i]\cdots Y[j+m_1-1,i])) \end{aligned}$$

(functions *hash* and REHASH are described in the section on the Karp–Rabin algorithm).

```
BF2D(X, m1, m2, Y, n1, n2)
  ▷ Searching
  for j1 ← 0 to n1 − m1
      do for j2 ← 0 to n2 − m2
             do i ← 0
                while i < m1 and x[i, 0 .. m2 − 1] = y[j1 + i, j2 .. j2 + m2 − 1]
                    do i ← i + 1
                if i ≥ m1
                  then OUTPUT(j1, j2)
```

FIGURE 13.22 The brute-force two-dimensional pattern matching algorithm.

```
KMP-IN-LINE(X, m1, m2, Y, n1, n2, x, y, next, j1)
i2 ← 0
j2 ← 0
while j2 < n2
    do while i2 > −1 and x[i2] ≠ y[j2]
            do i2 ← next[i2]
       i2 ← i2 + 1
       j2 ← j2 + 1
       if i2 ≥ m2
         then DIRECT-COMPARE(X, m1, m2, Y, n1, n2, j1, j2 − 1)
              i2 ← next[m2]
```

FIGURE 13.23 Search for x in y using KMP algorithm.

```
DIRECT-COMPARE(X, m1, m2, Y, row, column)
j1 ← row − m1 + 1
j2 ← column − m2 + 1
for i1 ← 0 to m1 − 1
    do for i2 ← 0 to m2 − 1
           do if X[i1, i2] ≠ Y[i1 + j1, i2 + j2]
                  then return
OUTPUT(j1, j2)
```

FIGURE 13.24 Naive check of an occurrence of x in y at position (*row, column*).

Example 13.9

$X =$

a	a	a
b	b	a
a	a	b

$Y =$

a	b	a	b	a	b	b
a	a	a	a	b	b	b
b	b	b	a	a	a	b
a	a	a	b	b	a	a
b	b	a	a	a	b	b
a	a	b	a	b	a	a

$x =$

681	681	680

$y =$

680	684	680	683	681	685	686

Next value of y is

681	681	681	680	684	683	685

. The occurrence of x at position 1 on y corresponds to an occurrence of X at position $(1, 1)$ on Y.

Since the alphabet of x and y is large, searching for x in y must be done by a string-matching algorithm for which the running time is independent of the size of the alphabet: the Knuth–Morris–Pratt suits this application perfectly. Its adaptation is shown in Figure 13.23.

When an occurrence of x is found in y, then we still have to check if an occurrence of X starts in Y at the corresponding position. This is done naively by the procedure of Figure 13.24.

The Zhu–Takaoka algorithm as explained above is displayed in Figure 13.25. The search for the pattern is performed row by row starting at row 0 and ending at row $n_1 - m_1$.

13.3.2 Bird/Baker Algorithm

The algorithm designed independently by Bird and Baker for the two-dimensional pattern matching problem combines the use of the Aho–Corasick algorithm and the Knuth–Morris–Pratt (KMP) algorithm.

```
ZT(X, m1, m2, Y, n1, n2)
▷ Preprocessing
▷ Computes x
for i2 ← 0 to m2 − 1
    do x[i2] ← 0
        for i1 ← 0 to m1 − 1
            do x[i2] ← (x[i2] << 1) + X[i1, i2]
▷ Computes the first value of y
for j2 ← 0 to n2 − 1
    do y[j2] ← 0
        for j1 ← 0 to m1 − 1
            do y[j2] ← (y[j2] << 1) + Y[j1, j2]
d ← 1
for i ← 1 to m1 − 1
    do d ← d << 1
next ← PREKMP(X′, m2)
▷ Searching
j1 ← m1 − 1
while j1 < n1
    do KMP-IN-LINE(X, m1, m2, Y, n1, n2, x, y, next, j2)
        if j1 < n1 − 1
          then for j2 ← 0 to n2 − 1
                    do y[j2] ← REHASH(Y[j1 − m1 + 1, j2], Y[j1 + 1, j2], y[j2])
        j1 ← j1 + 1
```

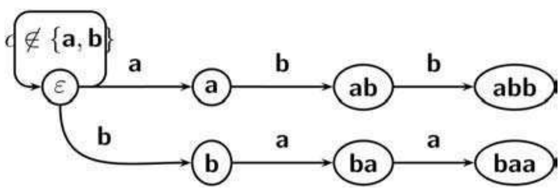
FIGURE 13.25 The Zhu-Takaoka two-dimensional pattern matching algorithm.

The pattern X is divided into its m_1 rows $R_0 = X[0, 0\,..\,m_2 - 1]$ to $R_{m_1-1} = x[m_1 - 1, 0\,..\,m_2 - 1]$. The rows are preprocessed into a trie as in the Aho–Corasick algorithm described earlier.

Example 13.10

Pattern X and the trie of its rows:

$$X = \begin{array}{|c|c|c|}\hline \mathbf{b} & \mathbf{a} & \mathbf{a} \\ \hline \mathbf{a} & \mathbf{b} & \mathbf{b} \\ \hline \mathbf{b} & \mathbf{a} & \mathbf{a} \\ \hline \end{array}$$



The search proceeds as follows. The text is read from the upper left corner to the bottom right corner, row by row. When reading the character $Y[j_1, j_2]$, the algorithm checks whether the portion $Y[j_1, j_2 - m_2 + 1\,..\,j_2] = R$ matches any of $R_0, \ldots, R_{m_1-1}$ using the Aho–Corasick machine. An additional one-dimensional array a of size n_1 is used as follows: $a[j_2] = k$ means that the $k - 1$ first rows $R_0, \ldots, R_{k-2}$ of the pattern match, respectively, the portions of the text: $Y[j_1 - k + 1, j_2 - m_2 + 1\,..\,j_2], \ldots, Y[j_1 - 1, j_2 - m_2 + 1\,..\,j_2]$. Then, if $R = R_{k-1}$, $a[j_2]$ is incremented to $k + 1$. If not, $a[j_2]$ is set to $s + 1$ where s is the maximum i such that

$$R_0 \cdots R_i = R_{k-s+1} \cdots R_{k-2} R.$$

```
PRE-KMP-FOR-B(X, m1, m2)
i ← 0
next[0] ← −1
j ← −1
while i < m1
    do while j > −1 and X[i, 0 . . m2 − 1] ≠ X[j, 0 . . m2 − 1]
            do j ← next[j]
        i ← i + 1
        j ← j + 1
        if X[i, 0 . . m2 − 1] ≠ X[j, 0 . . m2 − 1]
          then next[i] ← next[j]
          else next[i] ← j
return next
```

FIGURE 13.26 Computes the function *next* for rows of X.

```
B(X, m1, m2, Y, n1, n2)
▷ Preprocessing
for i ← 0 to m2 − 1
    do a[i] ← 0
root ← PREAC(m1)
next ← PRE-KMP-FOR-B(X, m1, m2)
for j1 ← 0 to n1 − 1
    do r ← root
       for j2 ← 0 to n2 − 1
           do while child(r, Y[j1, j2]) = UNDEFINED
                  do r ← fail(r)
              r ← child(r, Y[j1, j2])
              if out(r) ≠ ∅
                then k ← a[j2]
                     while k > 0 and X[k, 0 . . m2 − 1] = out(r)
                         do k ← next[k]
                     a[j2] ← k + 1
                     if k ≥ m1 − 1
                       then OUTPUT(j1 − m1 + 1, j2 − m2 + 1)
                else a[j2] ← 0
```

FIGURE 13.27 The Bird/Baker two-dimensional pattern matching algorithm.

The value s is computed using the KMP algorithm vertically (in columns). If there exists no such s, $a[j_2]$ is set to 0. Finally, if at some point $a[j_2] = m_1$, an occurrence of the pattern appears at position $(j_1 - m_1 + 1, j_2 - m_2 + 1)$ in the text.

The Bird/Baker algorithm is presented in Figure 13.26 and Figure 13.27. It runs in time $O((n_1 n_2 + m_1 m_2) \log \sigma)$.

13.4 Suffix Trees

The suffix tree $S(y)$ of a string y is a trie (as described earlier) containing all the suffixes of the string, and having the properties described subsequently. This data structure serves as an index on the string: it provides a direct access to all segments of the string, and gives the positions of all their occurrences in the string.

Once the suffix tree of a text y is built, searching for x in y remains to spell x along a branch of the tree. If this walk is successful, the positions of the pattern can be output. Otherwise, x does not occur in y.

```
SUFFIX-TREE(y, n)
T_{-1} ← one node tree
for j ← 0 to n − 1
    do T_j ← INSERT(T_{j−1}, y[j . . n − 1])
return T_{n−1}
```

FIGURE 13.28 Construction of a suffix tree for y.

```
INSERT(T_{j−1}, y[j . . n − 1])
1  locate the node h associated with head_j in T_{j−1}, possibly breaking an edge
2  add a new edge labeled tail_j from h to a new leaf representing y[j . . n − 1]
3  return the modified tree
```

FIGURE 13.29 Insertion of a new suffix in the tree.

Any kind of trie that represents the suffixes of a string can be used to search it. But the suffix tree has additional features which imply that its size is linear. The suffix tree of y is defined by the following properties:

- All branches of $S(y)$ are labeled by all suffixes of y.
- Edges of $S(y)$ are labeled by strings.
- Internal nodes of $S(y)$ have at least two children (when y is not empty).
- Edges outgoing an internal node are labeled by segments starting with different letters.
- The preceding segments are represented by their starting positions on y and their lengths.

Moreover, it is assumed that y ends with a symbol occurring nowhere else in it (the dollar sign is used in examples). This avoids marking nodes, and implies that $S(y)$ has exactly n leaves (number of nonempty suffixes). The other properties then imply that the total size of $S(y)$ is $O(n)$, which makes it possible to design a linear-time construction of the trie. The algorithm described in the present section has this time complexity provided the alphabet is fixed, or with an additional multiplicative factor $\log \sigma$ otherwise.

The algorithm inserts all nonempty suffixes of y in the data structure from the longest to the shortest suffix, as shown in Figure 13.28. We introduce two definitions to explain how the algorithm works:

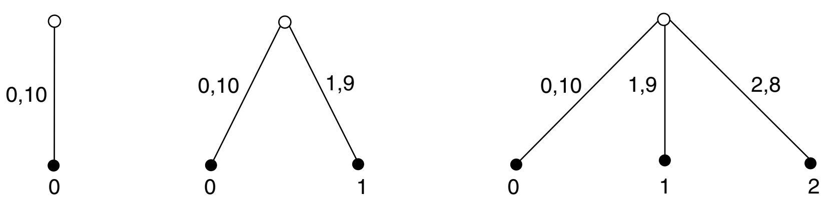
- $head_j$ is the longest prefix of $y[j \,..\, n-1]$ which is also a prefix of $y[i \,..\, n-1]$ for some $i < j$.
- $tail_j$ is the word such that $y[j \,..\, n-1] = head_j tail_j$.

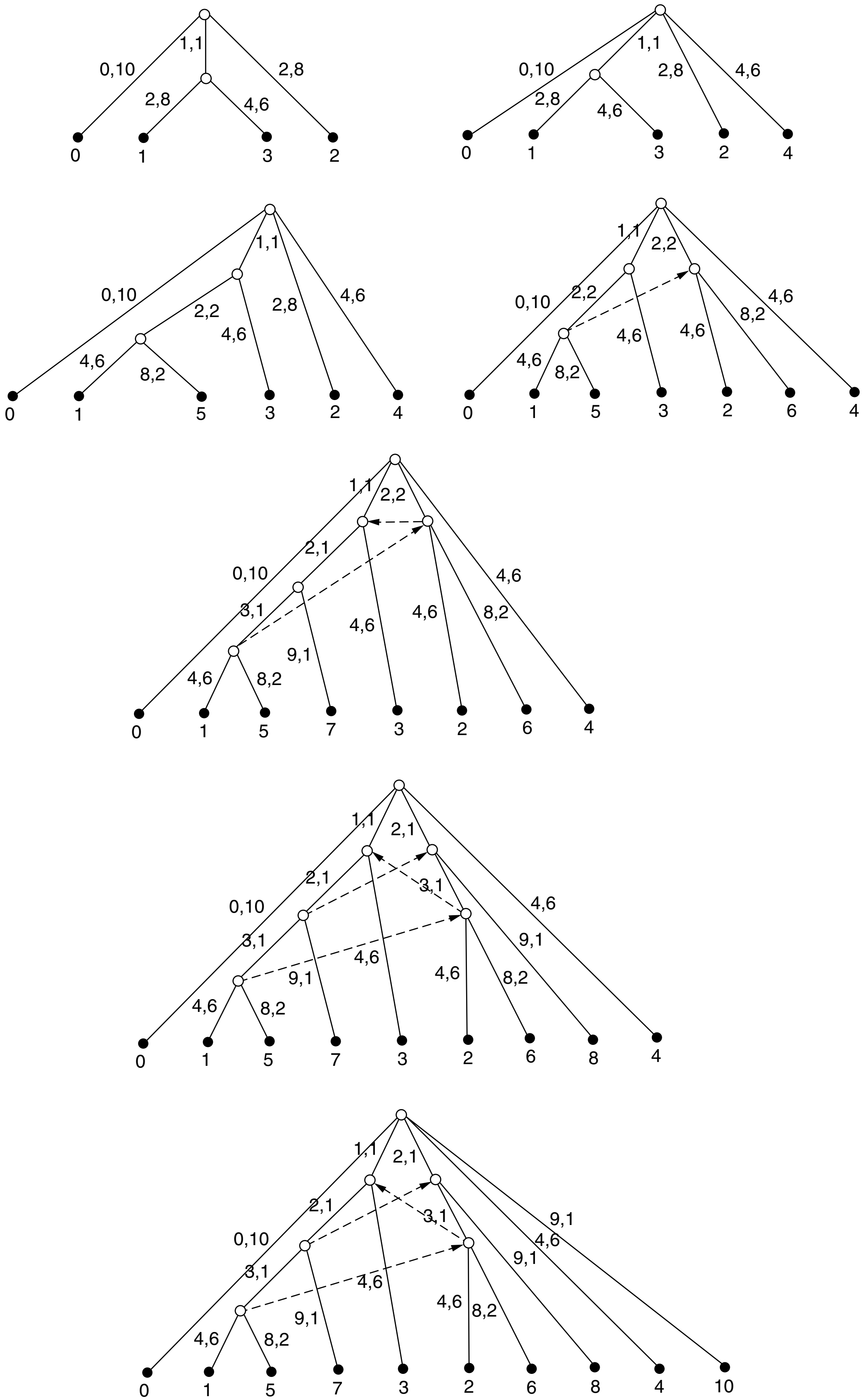
The strategy to insert the ith suffix in the tree is based on these definitions and described in Figure 13.29.

The second step of the insertion (Figure 13.29) is clearly performed in constant time. Thus, finding the node h is critical for the overall performance of the algorithm. A brute-force method to find it consists in spelling the current suffix $y[j \,..\, n-1]$ from the root of the tree, giving an $O(|head_j|)$ time complexity for the insertion at step j, and an $O(n^2)$ running time to build $S(y)$. Adding short-cut links leads to an overall $O(n)$ time complexity, although there is no guarantee that insertion at step j is realized in constant time.

Example 13.11

The different tries during the construction of the suffix tree of $y =$ **CAGATAGAG**. Leaves are black and labeled by the position of the suffix they represent. Plain arrows are labeled by pairs: the pair (j, ℓ) stands for the segment $y[j \,..\, j+\ell-1]$. Dashed arrows represent the nontrivial suffix links.




1,1
0,10
2,8
2,8
4,6
0
1
3
2
0,10
1,1
2,8
4,6
2,8
4,6
0
1
3
2
4
1,1
0,10
2,8
4,6
2,2
4,6
4,6
8,2
0
1
5
3
2
4
1,1
2,2
4,6
0,10
2,2
8,2
4,6
4,6
4,6
8,2
0
1
5
3
2
6
4
1,1
2,2
2,1
0,10
4,6
3,1
4,6
8,2
4,6
9,1
4,6
8,2
0
1
5
7
3
2
6
4
1,1
2,1
2,1
3,1
0,10
4,6
3,1
9,1
4,6
4,6
9,1
8,2
4,6
8,2
0
1
5
7
3
2
6
8
4
1,1
2,1
2,1
3,1
9,1
0,10
4,6
9,1
3,1
4,6
4,6
8,2
9,1
4,6
8,2
0
1
5
7
3
2
6
8
4
10

13.4.1 McCreight Algorithm

The key to get an efficient construction of the suffix tree $S(y)$ is to add links between nodes of the tree: they are called *suffix links*. Their definition relies on the relationship between $head_{j-1}$ and $head_j$: if $head_{j-1}$ is of the form az ($a \in \Sigma, z \in \Sigma^*$), then z is a prefix of $head_j$. In the suffix tree, the node associated with z is linked to the node associated with az. The suffix link creates a shortcut in the tree that helps with finding the next head efficiently. The insertion of the next suffix, namely, $head_j tail_j$, in the tree reduces to the insertion of $tail_j$ from the node associated with $head_j$.

The following property is an invariant of the construction: in T_j, only the node h associated with $head_j$ can fail to have a valid suffix link. This effectively happens when h has just been created at step j. The procedure to find the next head at step j is composed of two main phases:

A Rescanning: Assume that $head_{j-1} = az$ ($a \in \Sigma, z \in \Sigma^*$) and let d' be the associated node. If the suffix link on d' is defined, it leads to a node d from which the second step starts. Otherwise, the suffix link on d' is found by rescanning as follows. Let c' be the parent of d', and let (j, ℓ) be the label of edge (c', d'). For the ease of the description, assume that $az = av(y[j \,..\, j + \ell - 1])$ (it may happen that $az = y[j \,..\, j + \ell - 1]$). There is a suffix link defined on c' and going to some node c associated with v. The crucial observation here is that $y[j \,..\, j + \ell - 1]$ is the prefix of the label of some branch starting at node c. Then, the algorithm rescans $y[j \,..\, j + \ell - 1]$ in the tree: let e be the child of c along that branch, and let (k, m) be the label of edge (c, e). If $m < \ell$, then a recursive rescan of $q = y[j + m \,..\, j + \ell - 1]$ starts from node e. If $m > \ell$, the edge (c, e) is broken to insert a new node d; labels are updated correspondingly. If $m = \ell$, d is simply set to e. If the suffix link of d' is currently undefined, it is set to d.

B Scanning: A downward search starts from d to find the node h associated with $head_j$. The search is dictated by the characters of $tail_{j-1}$ one at a time from left to right. If necessary a new internal node is created at the end of the scanning.

After the two phases A and B are executed, the node associated with the new head is known, and the tail of the current suffix can be inserted in the tree.

To analyze the time complexity of the entire algorithm we mainly have to evaluate the total time of all scannings, and the total time of all rescannings. We assume that the alphabet is fixed, so that branching from a node to one of its children can be implemented to take constant time. Thus, the time spent for all scannings is linear because each letter of y is scanned only once. The same holds true for rescannings because each step downward (through node e) increases strictly the position of the segment of y considered there, and this position never decreases.

An implementation of McCreight's algorithm is shown in Figure 13.30. The next figures (Figure 13.31 through Figure 13.34) give the procedures used by the algorithm, especially procedures RESCAN and SCAN.

We use the following notation:

- *parent*(c) is the parent node of the node c
- *label*(c) is the pair (i, l) if the edge from the parent node of c to c itself is associated with the factor $y[i \,..\, i + l - 1]$
- *child*(c, a) is the only node that can be reached from the node c with the character a
- *link*(c) is the suffix node of the node c

13.5 Alignment

Alignments are used to compare strings. They are widely used in computational molecular biology. They constitute a mean to visualize resemblance between strings. They are based on notions of distance or similarity. Their computation is usually done by dynamic programming. A typical example of this method is the computation of the longest common subsequence of two strings. The reduction of the memory space presented on it can be applied to similar problems. We consider three different kinds of alignment of two

```
M(y, n)
root ← Init(y, n)
head ← root
tail ← child(root, y[0])
n ← n − 1
while n > 0
    do if head = root                                ▷ Phase A (rescanning)
          then d ← root
               (j, ℓ) ← label(tail)
               γ ← (j + 1, ℓ − 1)
          else γ ← label(tail)
               if link(head) ≠ UNDEFINED
                 then d ← link(head)
                 else (j, ℓ) ← label(head)
                      if parent(head) = root
                        then d ← Rescan(root, j + 1, ℓ − 1))
                        else d ← Rescan(link(parent(head)), j, ℓ))
                      link(head) ← d
       (head, γ) ← Scan(d, γ)                        ▷ Phase B (scanning)
       create a new node tail
       parent(tail) ← head
       label(tail) ← γ
       (j, ℓ) ← γ
       child(head, y[j]) ← tail
       n ← n − 1
return root
```

FIGURE 13.30 Suffix tree construction.

```
Init(y, n)
create a new node root
create a new node c
parent(root) ← UNDEFINED
parent(c) ← root
child(root, y[0]) ← c
label(root) ← UNDEFINED
label(c) ← (0, n)
return root
```

FIGURE 13.31 Initialization procedure.

```
Rescan(c, j, ℓ)
(k, m) ← label(child(c, y[j]))
while ℓ > 0 and ℓ ≥ m
    do c ← child(c, y[j])
       ℓ ← ℓ − m
       j ← j + m
       (k, m) ← label(child(c, y[j]))
if ℓ > 0
  then return Break-edge(child(c, y[j]), ℓ)
  else return c
```

FIGURE 13.32 The crucial rescan operation.

```
BREAK-EDGE(c, k)
create a new node g
parent(g) ← parent(c)
(j, ℓ) ← label(c)
child(parent(c), y[j]) ← g
label(g) ← (j, k)
parent(c) ← g
label(c) ← (j + k, ℓ − k)
child(g, y[j + k]) ← c
link(g) ← UNDEFINED
return g
```

FIGURE 13.33 Breaking an edge.

```
SCAN(d, γ)
(j, ℓ) ← γ
while child(d, y[j]) ≠ UNDEFINED
   do g ← child(d, y[j])
      k ← 1
      (s, lg) ← label(g)
      s ← s + 1
      ℓ ← ℓ − 1
      j ← j + 1
      while k < lg and y[j] = y[s]
         do j ← j + 1
            s ← s + 1
            k ← k + 1
            ℓ ← ℓ − 1
      if k < lg
        then return (BREAK-EDGE(g, k), (j, ℓ))
      d ← g
return (d, (j, ℓ))
```

FIGURE 13.34 The scan operation.

strings x and y: global alignment (that consider the whole strings x and y), local alignment (that enable to find the segment of x that is closer to a segment of y), and the longest common subsequence of x and y.

An **alignment** of two strings x and y of length m and n, respectively, consists in aligning their symbols on vertical lines. Formally, an alignment of two strings $x, y \in \Sigma$ is a word w on the alphabet $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus (\{(\varepsilon, \varepsilon)\}$ (ε is the empty word) whose projection on the first component is x and whose projection of the second component is y.

Thus, an alignment $w = (\overline{x}_0, \overline{y}_0)(\overline{x}_1, \overline{y}_1) \cdots (\overline{x}_{p-1}, \overline{y}_{p-1})$ of length p is such that $x = \overline{x}_0\overline{x}_1 \cdots \overline{x}_{p-1}$ and $y = \overline{y}_0\overline{y}_1 \cdots \overline{y}_{p-1}$ with $\overline{x}_i \in \Sigma \cup \{\varepsilon\}$ and $\overline{y}_i \in \Sigma \cup \{\varepsilon\}$ for $0 \leq i \leq p - 1$. The alignment is represented as follows

$$\begin{array}{cccc} \overline{x}_0 & \overline{x}_1 & \cdots & \overline{x}_{p-1} \\ \overline{y}_0 & \overline{y}_1 & \cdots & \overline{y}_{p-1} \end{array}$$

with the symbol $-$ instead of the symbol ε.

Example 13.12

A	**C**	**G**	–	–	**A**
A	**T**	**G**	**C**	**T**	**A**

is an alignment of **ACGA** and **ATGCTA**.

13.5.1 Global alignment

A global alignment of two strings x and y can be obtained by computing the distance between x and y. The notion of distance between two strings is widely used to compare files. The **diff** command of Unix operating system implements an algorithm based on this notion, in which lines of the files are treated as symbols. The output of a comparison made by **diff** gives the minimum number of operations (substitute a symbol, insert a symbol, or delete a symbol) to transform one file into the other.

Let us define the edit distance between two strings x and y as follows: it is the minimum number of elementary edit operations that enable to transform x into y. The elementary edit operations are:

- The substitution of a character of x at a given position by a character of y
- The deletion of a character of x at a given position
- The insertion of a character of y in x at a given position

A cost is associated to each elementary edit operation. For $a, b \in \Sigma$:

- $Sub(a, b)$ denotes the cost of the substitution of the character a by the character b,
- $Del(a)$ denotes the cost of the deletion of the character a, and
- $Ins(a)$ denotes the cost of the insertion of the character a.

This means that the costs of the edit operations are independent of the positions where the operations occur. We can now define the edit distance of two strings x and y by

$$edit(x, y) = \min\{\text{cost of } \gamma \mid \gamma \in \Gamma_{x,y}\}$$

where $\Gamma_{x,y}$ is the set of all the sequences of edit operations that transform x into y, and the cost of an element $\gamma \in \Gamma_{x,y}$ is the sum of the costs of its elementary edit operations.

To compute $edit(x, y)$ for two strings x and y of length m and n, respectively, we make use of a two-dimensional table T of $m + 1$ rows and $n + 1$ columns such that

$$T[i, j] = edit(x[i], y[j])$$

for $i = 0, \ldots, m - 1$ and $j = 0, \ldots, n - 1$. It follows $edit(x, y) = T[m - 1, n - 1]$.

The values of the table T can be computed by the following recurrence formula:

$$\begin{aligned}
T[-1, -1] &= 0 \\
T[i, -1] &= T[i - 1, -1] + Del(x[i]) \\
T[-1, j] &= T[-1, j - 1] + Ins(y[j]) \\
T[i, j] &= \min \begin{cases} T[i - 1, j - 1] + Sub(x[i], y[j]) \\ T[i - 1, j] + Del(x[i]) \\ T[i, j - 1] + Ins(y[j]) \end{cases}
\end{aligned}$$

for $i = 0, 1, \ldots, m - 1$ and $j = 0, 1, \ldots, n - 1$.

```
GENERIC-DP(x, m, y, n, MARGIN, FORMULA)
MARGIN(T, x, m, y, n)
for j ← 0 to n − 1
    do for i ← 0 to m − 1
            do T[i, j] ← FORMULA(T, x, i, y, j)
return T
```

FIGURE 13.35 Computation of the table T by dynamic programming.

```
MARGIN-GLOBAL(T, x, m, y, n)
T[−1, −1] ← 0
for i ← 0 to m − 1
    do T[i, −1] ← T[i − 1, −1] + Del(x[i])
for j ← 0 to n − 1
    do T[−1, j] ← T[−1, j − 1] + Ins(y[j])
```

FIGURE 13.36 Margin initialization for the computation of a global alignment.

FORMULA-GLOBAL(T, x, i, y, j)

$$\textbf{return}\ \min \begin{cases} T[i-1, j-1] + Sub(x[i], y[j]) \\ T[i-1, j] + Del(x[i]) \\ T[i, j-1] + Ins(y[j]) \end{cases}$$

FIGURE 13.37 Computation of $T[i, j]$ for a global alignment.

The value at position (i, j) in the table T only depends on the values at the three neighbor positions $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$.

The direct application of the above recurrence formula gives an exponential time algorithm to compute $T[m-1, n-1]$. However, the whole table T can be computed in quadratic time technique known as "dynamic programming." This is a general technique that is used to solve the different kinds of alignments.

The computation of the table T proceeds in two steps. First it initializes the first column and first row of T; this is done by a call to a generic function MARGIN, which is a parameter of the algorithm and that depends on the kind of alignment considered. Second, it computes the remaining values of T, which is done by a call to a generic function FORMULA, which is a parameter of the algorithm and that depends on the kind of alignment considered. Computing a global alignment of x and y can be done by a call to GENERIC-DP with the following parameters $(x, m, y, n,$ MARGIN-GLOBAL, FORMULA-GLOBAL$)$ (see Figure 13.35, Figure 13.36, and Figure 13.37). The computation of all the values of the table T can thus be done in quadratic space and time: $O(m \times n)$.

An optimal alignment (with minimal cost) can then be produced by a call to the function ONE-ALIGNMENT$(T, x, m-1, y, n-1)$ (see Figure 13.38). It consists in tracing back the computation of the values of the table T from position $[m-1, n-1]$ to position $[-1, -1]$. At each cell $[i, j]$, the algorithm determines among the three values $T[i-1, j-1]+Sub(x[i], y[j])$, $T[i-1, j]+Del(x[i])$, and $T[i, j-1]+Ins(y[j])$) which has been used to produce the value of $T[i, j]$. If $T[i-1, j-1] + Sub(x[i], y[j])$ has been used it adds $(x[i], y[j])$ to the optimal alignment and proceeds recursively with the cell at $[i-1, j-1]$. If $T[i-1, j] + Del(x[i])$ has been used, it adds $(x[i], -)$ to the optimal alignment and proceeds recursively with cell at $[i-1, j]$. If $T[i, j-1] + Ins(y[j])$ has been used, it adds $(-, y[j])$ to the optimal alignment

```
ONE-ALIGNMENT(T, x, i, y, j)
if i = −1 and j = −1
  then return (ε, ε)
  else if i = −1
         then return ONE-ALIGNMENT(T, x, −1, y, j − 1) · (ε, y[j])
       elseif j = −1
         then return ONE-ALIGNMENT(T, x, i − 1, y, −1) · (x[i], ε)
       else    if T[i, j] = T[i − 1, j − 1] + Sub(x[i], y[j])
                 then return ONE-ALIGNMENT(T, x, i − 1, y, j − 1) · (x[i], y[j])
               elseif T[i, j] = T[i − 1, j] + Del(x[i])
                 then return ONE-ALIGNMENT(T, x, i − 1, y, j) · (x[i], ε)
               else    return ONE-ALIGNMENT(T, x, i, y, j − 1) · (ε, y[j])
```

FIGURE 13.38 Recovering an optimal alignment.

and proceeds recursively with cell at $[i, j-1]$. Recovering all the optimal alignments can be done by a similar technique.

Example 13.13

T	j	-1	0	1	2	3	4	5
i		$y[j]$	**A**	**T**	**G**	**C**	**T**	**A**
-1	$x[i]$	0	1	2	3	4	5	6
0	**A**	1	0	1	2	3	4	5
1	**C**	2	1	1	2	2	3	4
2	**G**	3	2	2	1	2	3	4
3	**A**	4	3	3	2	2	3	3

The values of the above table have been obtained with the following unitary costs: $Sub(a, b) = 1$ if $a \neq b$ and $Sub(a, a) = 0$, $Del(a) = Ins(a) = 1$ for $a, b \in \Sigma$.

13.5.2 Local Alignment

A local alignment of two strings x and y consists in finding the segment of x that is closer to a segment of y. The notion of distance used to compute global alignments cannot be used in that case because the segments of x closer to segments of y would only be the empty segment or individual characters. This is why a notion of similarity is used based on a scoring scheme for edit operations.

A score (instead of a cost) is associated to each elementary edit operation. For $a, b \in \Sigma$:

- $Sub_S(a, b)$ denotes the score of substituting the character b for the character a.
- $Del_S(a)$ denotes the score of deleting the character a.
- $Ins_S(a)$ denotes the score of inserting the character a.

This means that the scores of the edit operations are independent of the positions where the operations occur. For two characters a and b, a positive value of $Sub_S(a, b)$ means that the two characters are close to each other, and a negative value of $Sub_S(a, b)$ means that the two characters are far apart.

We can now define the edit score of two strings x and y by

$$sco(x, y) = \max\{\text{score of } \gamma \mid \gamma \in \Gamma_{x,y}\}$$

where $\Gamma_{x,y}$ is the set of all the sequences of edit operations that transform x into y and the score of an element $\sigma \in \Gamma_{x,y}$ is the sum of the scores of its elementary edit operations.

To compute $sco(x, y)$ for two strings x and y of length m and n, respectively, we make use of a two-dimensional table T of $m + 1$ rows and $n + 1$ columns such that

$$T[i, j] = sco(x[i], y[j])$$

for $i = 0, \ldots, m - 1$ and $j = 0, \ldots, n - 1$. Therefore, $sco(x, y) = T[m - 1, n - 1]$.

The values of the table T can be computed by the following recurrence formula:

$$\begin{aligned}
T[-1, -1] &= 0\,,\\
T[i, -1] &= 0\,,\\
T[-1, j] &= 0\,,\\
T[i, j] &= \max \begin{cases} T[i-1, j-1] + Sub_S(x[i], y[j])\,,\\ T[i-1, j] + Del_S(x[i])\,,\\ T[i, j-1] + Ins_S(y[j])\,,\\ 0\,, \end{cases}
\end{aligned}$$

for $i = 0, 1, \ldots, m - 1$ and $j = 0, 1, \ldots, n - 1$.

Computing the values of T for a local alignment of x and y can be done by a call to GENERIC-DP with the following parameters $(x, m, y, n,$ MARGIN-LOCAL, FORMULA-LOCAL$)$ in $O(mn)$ time and space complexity (see Figure 13.35, Figure 13.39, and Figure 13.40). Recovering a local alignment can be done in a way similar to what is done in the case of a global alignment (see Figure 13.38) but the trace back procedure must start at a position of a maximal value in T rather than at position $[m - 1, n - 1]$.

MARGIN-LOCAL(T, x, m, y, n)

$T[-1, -1] \leftarrow 0$
for $i \leftarrow 0$ **to** $m - 1$
  do $T[i, -1] \leftarrow 0$
for $j \leftarrow 0$ **to** $n - 1$
  do $T[-1, j] \leftarrow 0$

FIGURE 13.39 Margin initialization for computing a local alignment.

FORMULA-LOCAL(T, x, i, y, j)

return $\max \begin{cases} T[i-1, j-1] + Sub_S(x[i], y[j])\\ T[i-1, j] + Del_S(x[i])\\ T[i, j-1] + Ins_S(y[j])\\ 0 \end{cases}$

FIGURE 13.40 Recurrence formula for computing a local alignment.

Example 13.14

Computation of an optimal local alignment of x = **EAWACQGKL** and y = **ERDAWCQPGKWY** with scores:

$Sub_S(a,a) = 1$, $Sub_S(a,b) = -3$ and $Del_S(a) = Ins_S(a) = -1$ for $a, b \in \Sigma$, $a \neq b$.

T	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	**E**	**R**	**D**	**A**	**W**	**C**	**Q**	**P**	**G**	**K**	**W**	**Y**
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	**E**	0	1	0	0	0	0	0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	0	0	0	0	0	0	0	0
2	**W**	0	0	0	0	0	2	1	0	0	0	0	1	0
3	**A**	0	0	0	0	1	1	0	0	0	0	0	0	0
4	**C**	0	0	0	0	0	0	2	1	0	0	0	0	0
5	**Q**	0	0	0	0	0	0	1	3	2	1	0	0	0
6	**G**	0	0	0	0	0	0	0	2	1	3	2	1	0
7	**K**	0	0	0	0	0	0	0	1	0	2	4	3	2
8	**L**	0	0	0	0	0	0	0	0	0	1	3	2	1

The corresponding optimal local alignment is:

A	**W**	**A**	**C**	**Q**	-	**G**	**K**
A	**W**	-	**C**	**Q**	**P**	**G**	**K**

13.5.3 Longest Common Subsequence of Two Strings

A subsequence of a word x is obtained by deleting zero or more characters from x. More formally, $w[0\,..\,i-1]$ is a subsequence of $x[0\,..\,m-1]$ if there exists an increasing sequence of integers $(k_j \mid j = 0, \ldots, i-1)$ such that for $0 \leq j \leq i-1$, $w[j] = x[k_j]$. We say that a word is an $\mathrm{lcs}(x, y)$ if it is a **longest common subsequence** of the two words x and y. Note that two strings can have several longest common subsequences. Their common length is denoted by $\mathrm{llcs}(x, y)$.

A brute-force method to compute an $\mathrm{lcs}(x, y)$ would consist in computing all the subsequences of x, checking if they are subsequences of y, and keeping the longest one. The word x of length m has 2^m subsequences, and so this method could take $O(2^m)$ time, which is impractical even for fairly small values of m.

However, $\mathrm{llcs}(x, y)$ can be computed with a two-dimensional table T by the following recurrence formula:

$$
\begin{aligned}
T[-1,-1] &= 0\,,\\
T[i,-1] &= 0\,,\\
T[-1,j] &= 0\,,\\
T[i,j] &= \begin{cases} T[i-1,j-1]+1 & \text{if } x[i] = y[j],\\ \max(T[i-1,j], T[i,j-1]) & \text{otherwise,}\end{cases}
\end{aligned}
$$

for $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$. Then, $T[i, j] = \mathrm{llcs}(x[0\,..\,i], y[0\,..\,j])$ and $\mathrm{llcs}(x, y) = T[m-1, n-1]$.

Computing $T[m-1, n-1]$ can be done by a call to GENERIC-DP with the following parameters $(x, m, y, n,$ MARGIN-LOCAL, FORMULA-LCS$)$ in $O(mn)$ time and space complexity (see Figure 13.35, Figure 13.39, and Figure 13.41).

```
FORMULA-LCS(T, x, i, y, j)
if x[i] = y[j]
   then return T[i − 1, j − 1] + 1
   else return max{T[i − 1, j], T[i, j − 1]}
```

FIGURE 13.41 Recurrence formula for computing an *lcs*.

It is possible afterward to trace back a path from position $[m-1, n-1]$ in order to exhibit an $\text{lcs}(x, y)$ in a similar way as for producing a global alignment (see Figure 13.38).

Example 13.15

The value $T[4, 8] = 4$ is $\text{llcs}(x, y)$ for $x =$ **AGCGA** and $y =$ **CAGATAGAG**. String **AGGA** is an lcs of x and y.

T	j	-1	0	1	2	3	4	5	6	7	8
i		$y[j]$	**C**	**A**	**G**	**A**	**T**	**A**	**G**	**A**	**G**
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0
0	**A**	0	0	1	1	1	1	1	1	1	1
1	**G**	0	0	1	2	2	2	2	2	2	2
2	**C**	0	1	1	2	2	2	2	2	2	2
3	**G**	0	1	1	2	2	2	2	3	3	3
4	**A**	0	1	2	2	3	3	3	3	4	4

13.5.4 Reducing the Space: Hirschberg Algorithm

If only the length of an $\text{lcs}(x, y)$ is required, it is easy to see that only one row (or one column) of the table T needs to be stored during the computation. The space complexity becomes $O(\min(m, n))$, as can be checked on the algorithm of Figure 13.42. Indeed, the Hirschberg algorithm computes an $\text{lcs}(x, y)$ in linear space and not only the value $\text{llcs}(x, y)$. The computation uses the algorithm of Figure 13.43.

Let us define

$$T^*[i, n] = T^*[m, j] = 0, \quad \text{for } 0 \le i \le m \quad \text{and} \quad 0 \le j \le n$$

$$T^*[m-i, n-j] = \text{llcs}((x[i\,..\,m-1])^R, (y[j\,..\,n-1])^R)$$

$$\text{for } 0 \le i \le m-1 \quad \text{and} \quad 0 \le j \le n-1$$

and

$$M(i) = \max_{0 \le j < n} \{T[i, j] + T^*[m-i, n-j]\}$$

where the word w^R is the reverse (or mirror image) of the word w. The following property is the key observation to compute an $\text{lcs}(x, y)$ in linear space:

$$M(i) = T[m-1, n-1], \quad \text{for } 0 \le i < m.$$

In the algorithm shown in Figure 13.43, the integer j is chosen as $n/2$. After $T[i, j-1]$ and $T^*[m-i, n-j]$ $(0 \le i < m)$ are computed, the algorithm finds an integer k such that $T[i, k] + T^*[m-i, n-k] = T[m-1, n-1]$. Then, recursively, it computes an $\text{lcs}(x[0\,..\,k-1], y[0\,..\,j-1])$ and an $\text{lcs}(x[k\,..\,m-1], y[j\,..\,n-1])$, and concatenates them to get an $\text{lcs}(x, y)$.

```
LLCS(x, m, y, n)
for i ← −1 to m − 1
    do C[i] ← 0
for j ← 0 to n − 1
    do last ← 0
       for i ← −1 to m − 1
           do if last > C[i]
                 then C[i] ← last
               elseif last < C[i]
                 then last ← C[i]
               elseif x[i] = y[j]
                 then C[i] ← C[i] + 1
                      last ← last + 1
return C
```

FIGURE 13.42 $O(m)$-space algorithm to compute llcs(x, y).

```
HIRSCHBERG(x, m, y, n)
if m = 0
  then return ε
  else if m = 1
         then if x[0] ∈ y
                then return x[0]
                else return ε
         else j ← ⌊n/2⌋
              C ← LLCS(x, m, y[0 .. j − 1], j)
              C* ← LLCS(x^R, m, y[j .. n − 1]^R, n − j)
              k ← m − 1
              M ← C[m − 1] + C*[m − 1]
              for j ← −1 to m − 2
                  do if C[j] + C*[j] > M
                        then M ← C[j] + C*[j]
                             k ← j
              return HIRSCHBERG(x[0 .. k − 1], k, y[0 .. j − 1], j)·
                     HIRSCHBERG(x[k .. m − 1], m − k, y[j .. n − 1], n − j)
```

FIGURE 13.43 $O(\min(m, n))$-space computation of lcs(x, y).

The running time of the Hirschberg algorithm is still $O(mn)$ but the amount of space required for the computation becomes $O(\min(m, n))$, instead of being quadratic when computed by dynamic programming.

13.6 Approximate String Matching

Approximate string matching is the problem of finding all approximate occurrences of a pattern x of length m in a text y of length n. Approximate occurrences of x are segments of y that are close to x according to a specific distance: the distance between segments and x must be not greater than a given integer k. We consider two distances in this section: the **Hamming distance** and the **Levenshtein distance**.

With the Hamming distance, the problem is also known as approximate string matching with k mismatches. With the Levenshtein distance (or edit distance), the problem is known as approximate string matching with k differences.

The Hamming distance between two words w_1 and w_2 of the same length is the number of positions with different characters. The Levenshtein distance between two words w_1 and w_2 (not necessarily of the same length) is the minimal number of differences between the two words. A difference is one of the following operations:

- A substitution: a character of w_1 corresponds to a different character in w_2.
- An insertion: a character of w_1 corresponds to no character in w_2.
- A deletion: a character of w_2 corresponds to no character in w_1.

The *Shift-Or algorithm* of the next section is a method that is both very fast in practice and very easy to implement. It solves the Hamming distance and the Levenshtein distance problems. We initially describe the method for the exact string-matching problem and then show how it can handle the cases of k mismatches and k differences. The method is flexible enough to be adapted to a wide range of similar approximate matching problems.

13.6.1 Shift-Or Algorithm

We first present an algorithm to solve the exact string-matching problem using a technique different from those developed previously, but which extends readily to the approximate string-matching problem.

Let $\mathbf{R}^0$ be a bit array of size m. Vector $\mathbf{R}^0_j$ is the value of the entire array $\mathbf{R}^0$ after text character $y[j]$ has been processed (see Figure 13.44). It contains information about all matches of prefixes of x that end at position j in the text. It is defined, for $0 \le i \le m-1$, by

$$\mathbf{R}^0_j[i] = \begin{cases} 0 & \text{if } x[0\,..\,i] = y[j-i\,..\,j] \\ 1 & \text{otherwise.} \end{cases}$$

Therefore, $\mathbf{R}^0_j[m-1] = 0$ is equivalent to saying that an (exact) occurrence of the pattern x ends at position j in y.

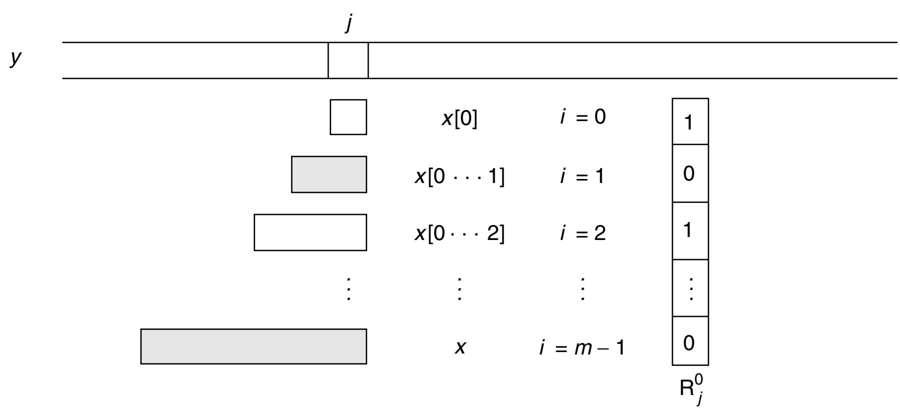


FIGURE 13.44 Meaning of vector $\mathbf{R}^0_j$.

The vector $\mathbf{R}_j^0$ can be computed after $\mathbf{R}_{j-1}^0$ by the following recurrence relation:

$$\mathbf{R}_j^0[i] = \begin{cases} 0 & \text{if } \mathbf{R}_{j-1}^0[i-1] = 0 \text{ and } x[i] = y[j], \\ 1 & \text{otherwise,} \end{cases}$$

and

$$\mathbf{R}_j^0[0] = \begin{cases} 0 & \text{if } x[0] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The transition from $\mathbf{R}_{j-1}^0$ to $\mathbf{R}_j^0$ can be computed very fast as follows. For each $a \in \Sigma$, let S_a be a bit array of size m defined, for $0 \leq i \leq m-1$, by

$$S_a[i] = 0 \quad \text{if} \quad x[i] = a.$$

The array S_a denotes the positions of the character a in the pattern x. All arrays S_a are preprocessed before the search starts. And the computation of $\mathbf{R}_j^0$ reduces to two operations, SHIFT and OR:

$$\mathbf{R}_j^0 = \text{SHIFT}(\mathbf{R}_{j-1}^0) \quad \text{OR} \quad S_{y[j]}.$$

Example 13.16

String $x =$ **GATAA** occurs at position 2 in $y =$ **CAGATAAGAGAA**.

S_{A}	S_{C}	S_{G}	S_{T}
1	1	0	1
0	1	1	1
1	1	1	0
0	1	1	1
0	1	1	1

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	1	1	0	1	1	1	1	0	1	0	1	1
A	1	1	1	0	1	1	1	1	0	1	0	1
T	1	1	1	1	0	1	1	1	1	1	1	1
A	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1

13.6.2 String Matching with k Mismatches

The Shift-Or algorithm easily adapts to support approximate string matching with k mismatches. To simplify the description, we shall present the case where at most one substitution is allowed.

We use arrays $\mathbf{R}^0$ and S as before, and an additional bit array $\mathbf{R}^1$ of size m. Vector $\mathbf{R}_{j-1}^1$ indicates all matches with at most one substitution up to the text character $y[j-1]$. The recurrence on which the computation is based splits into two cases.

1. There is an exact match on the first i characters of x up to $y[j-1]$ (i.e., $\mathbf{R}_{j-1}^0[i-1] = 0$). Then, substituting $x[i]$ to $y[j]$ creates a match with one substitution (see Figure 13.45). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^0[i-1]\,.$$

2. There is a match with one substitution on the first i characters of x up to $y[j-1]$ and $x[i] = y[j]$. Then, there is a match with one substitution of the first $i+1$ characters of x up to $y[j]$

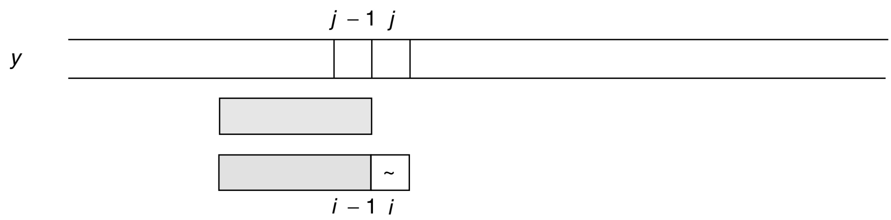


FIGURE 13.45 If $\mathbf{R}^0_{j-1}[i-1] = 0$, then $\mathbf{R}^1_j[i] = 0$.

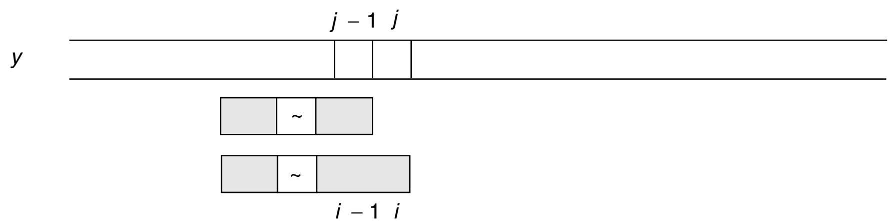


FIGURE 13.46 $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

(see Figure 13.46). Thus,

$$\mathbf{R}^1_j[i] = \begin{cases} \mathbf{R}^1_{j-1}[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This implies that $\mathbf{R}^1_j$ can be updated from $\mathbf{R}^1_{j-1}$ by the relation:

$$\mathbf{R}^1_j = \left(\text{SHIFT}\left(\mathbf{R}^1_{j-1}\right) \quad \text{OR} \quad S_{y[j]}\right) \quad \text{AND} \quad \text{SHIFT}\left(\mathbf{R}^0_{j-1}\right).$$

Example 13.17

String x = **GATAA** occurs at positions 2 and 7 in y = **CAGATAAGAGAA** with no more than one mismatch.

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	0	1	0	0	1	0	1	0	0
T	1	1	1	1	0	1	1	1	1	0	1	0
A	1	1	1	1	1	0	1	1	1	1	0	1
A	1	1	1	1	1	1	0	1	1	1	1	0

13.6.3 String Matching with *k* Differences

We show in this section how to adapt the Shift-Or algorithm to the case of only one insertion, and then dually to the case of only one deletion. The method is based on the following elements.

One insertion is allowed: here, vector $\mathbf{R}^1_{j-1}$ indicates all matches with at most one insertion up to text character $y[j-1]$. $\mathbf{R}^1_{j-1}[i-1] = 0$ if the first i characters of x ($x[0 \,..\, i-1]$) match i symbols of the last $i+1$ text characters up to $y[j-1]$. Array $\mathbf{R}^0$ is maintained as before, and we show how to maintain array $\mathbf{R}^1$. Two cases arise.

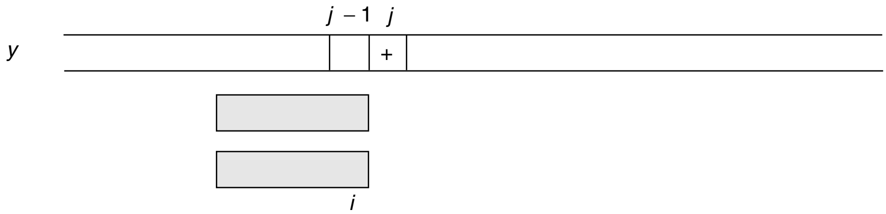


FIGURE 13.47 If $\mathbf{R}^0_{j-1}[i] = 0$, then $\mathbf{R}^1_j[i] = 0$.

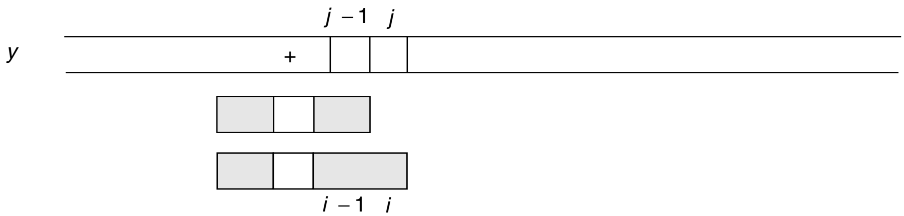


FIGURE 13.48 $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

1. There is an exact match on the first $i+1$ characters of x ($x[0\,..\,i]$) up to $y[j-1]$. Then inserting $y[j]$ creates a match with one insertion up to $y[j]$ (see Figure 13.47). Thus,

$$\mathbf{R}^1_j[i] = \mathbf{R}^0_{j-1}[i]\,.$$

2. There is a match with one insertion on the i first characters of x up to $y[j-1]$. Then if $x[i] = y[j]$, there is a match with one insertion on the first $i+1$ characters of x up to $y[j]$ (see Figure 13.48). Thus,

$$\mathbf{R}^1_j[i] = \begin{cases} \mathbf{R}^1_{j-1}[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This shows that $\mathbf{R}^1_j$ can be updated from $\mathbf{R}^1_{j-1}$ with the formula

$$\mathbf{R}^1_j = \left(\text{SHIFT}\left(\mathbf{R}^1_{j-1}\right) \quad \text{OR} \quad S_{y[j]}\right) \quad \text{AND} \quad \mathbf{R}^0_{j-1}\,.$$

Example 13.18

Here, **GATAAG** is an occurrence of $x =$ **GATAA** with exactly one insertion in $y =$ **CAGATAAGAGAA**

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	1	1	1	0	1	1	1	1	0	1	0	1
A	1	1	1	1	0	1	1	1	1	0	1	0
T	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1
A	1	1	1	1	1	1	1	0	1	1	1	1

One deletion is allowed: we assume here that $\mathbf{R}^1_{j-1}$ indicates all possible matches with at most one deletion up to $y[j-1]$. As in the previous solution, two cases arise.

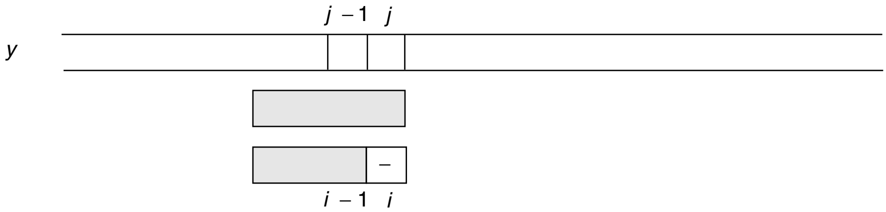


FIGURE 13.49 If $\mathbf{R}_j^0[i] = 0$, then $\mathbf{R}_j^1[i] = 0$.

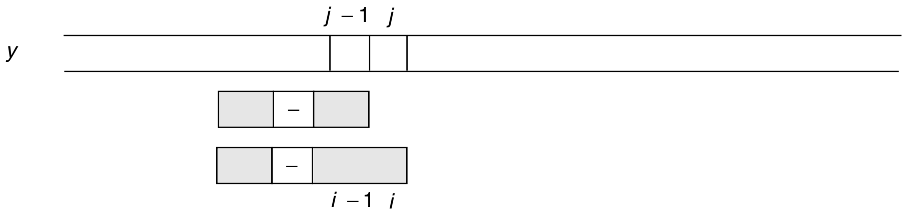


FIGURE 13.50 $\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^1[i-1]$ if $x[i] = y[j]$.

1. There is an exact match on the first $i+1$ characters of x ($x[0\,..\,i]$) up to $y[j]$ (i.e., $\mathbf{R}_j^0[i] = 0$). Then, deleting $x[i]$ creates a match with one deletion (see Figure 13.49). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_j^0[i].$$

2. There is a match with one deletion on the first i characters of x up to $y[j-1]$ and $x[i] = y[j]$. Then, there is a match with one deletion on the first $i+1$ characters of x up to $y[j]$ (see Figure 13.50). Thus,

$$\mathbf{R}_j^1[i] = \begin{cases} \mathbf{R}_{j-1}^1[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The discussion provides the following formula used to update $\mathbf{R}_j^1$ from $\mathbf{R}_{j-1}^1$:

$$\mathbf{R}_j^1 = \big(\text{SHIFT}(\mathbf{R}_{j-1}^1) \quad \text{OR} \quad S_{y[j]}\big) \quad \text{AND} \quad \text{SHIFT}(\mathbf{R}_j^0).$$

Example 13.19

GATA and **ATAA** are two occurrences with one deletion of $x =$ **GATAA** in $y =$ **CAGATAAGAGAA**

	C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	1	0	0	0	0	0	0	0
T	1	1	1	0	0	1	1	1	0	1	0	1
A	1	1	1	1	0	0	1	1	1	1	1	0
A	1	1	1	1	1	0	0	1	1	1	1	1

13.6.4 Wu–Manber Algorithm

We present in this section a general solution for the approximate string-matching problem with at most k differences of the types: insertion, deletion, and substitution. It is an extension of the problems presented

above. The following algorithm maintains $k+1$ bit arrays $\mathbf{R}^0, \mathbf{R}^1, \ldots, \mathbf{R}^k$ that are described now. The vector $\mathbf{R}^0$ is maintained similarly as in the exact matching case (Section 13.6.1). The other vectors are computed with the formula ($1 \leq \ell \leq k$)

$$\begin{aligned} \mathbf{R}^\ell_j = \big(&\mathrm{SHIFT}\big(\mathbf{R}^\ell_{j-1}\big) \quad \mathrm{OR} \quad S_{y[j]}\big) \\ &\mathrm{AND} \quad \mathrm{SHIFT}\big(\mathbf{R}^{\ell-1}_j\big) \\ &\mathrm{AND} \quad \mathrm{SHIFT}\big(\mathbf{R}^{\ell-1}_{j-1}\big) \\ &\mathrm{AND} \quad \mathbf{R}^{\ell-1}_{j-1} \end{aligned}$$

which can be rewritten into

$$\begin{aligned} \mathbf{R}^\ell_j = \big(&\mathrm{SHIFT}\big(\mathbf{R}^\ell_{j-1}\big) \quad \mathrm{OR} \quad S_{y[j]}\big) \\ &\mathrm{AND} \quad \mathrm{SHIFT}\big(\mathbf{R}^{\ell-1}_j \quad \mathrm{AND} \quad \mathbf{R}^{\ell-1}_{j-1}\big) \\ &\mathrm{AND} \quad \mathbf{R}^{\ell-1}_{j-1}. \end{aligned}$$

Example 13.20

Here, $x =$ **GATAA** and $y =$ **CAGATAAGAGAA** and $k = 1$. The output 5, 6, 7, and 11 corresponds to the segments **GATA**, **GATAA**, **GATAAG**, and **GAGAA**, which approximate the pattern **GATAA** with no more than one difference.

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	0	0	0	0	0	0
T	1	1	1	0	0	0	1	1	0	0	0	0
A	1	1	1	1	0	0	0	1	1	1	0	0
A	1	1	1	1	1	0	0	0	1	1	1	0

The method, called the Wu–Manber algorithm, is implemented in Figure 13.51. It assumes that the length of the pattern is no more than the size of the memory word of the machine, which is often the case in applications.

```
WM(x, m, y, n, k)
for each character a ∈ Σ
    do S_a ← 1^m
for i ← 0 to m − 1
    do S_x[i][i] ← 0
R^0 ← 1^m
for ℓ ← 1 to k
    do R^ℓ ← SHIFT(R^(ℓ−1))
for j ← 0 to n − 1
    do T ← R^0
       R^0 ← SHIFT(R^0)  OR  S_y[j]
       for ℓ ← 1 to k
           do T' ← R^ℓ
              R^ℓ ← (SHIFT(R^ℓ)  OR  S_y[j])  AND  (SHIFT((T  AND  R^(ℓ−1)))  AND  T
              T ← T'
       if R^k[m − 1] = 0
         then OUTPUT(j)
```

FIGURE 13.51 Wu–Manber approximate string-matching algorithm.

The preprocessing phase of the algorithm takes $O(\sigma m + km)$ memory space, and runs in time $O(\sigma m + k)$. The time complexity of its searching phase is $O(kn)$.

13.7 Text Compression

In this section we are interested in algorithms that compress texts. Compression serves both to save storage space and to save transmission time. We shall assume that the uncompressed text is stored in a file. The aim of compression algorithms is to produce another file containing the compressed version of the same text. Methods in this section work with no loss of information, so that decompressing the compressed text restores exactly the original text.

We apply two main strategies to design the algorithms. The first strategy is a statistical method that takes into account the frequencies of symbols to build a uniquely decipherable code optimal with respect to the compression. The code contains new codewords for the symbols occurring in the text. In this method, fixed-length blocks of bits are encoded by different codewords. *A contrario*, the second strategy encodes variable-length segments of the text. To put it simply, the algorithm, while scanning the text, replaces some already read segments just by a pointer to their first occurrences.

Text compression software often use a mixture of several methods. An example of that is given in Section 13.7.3, which contains in particular two classical simple compression algorithms. They compress efficiently only a small variety of texts when used alone, but they become more powerful with the special preprocessing presented there.

13.7.1 Huffman Coding

The Huffman method is an optimal statistical coding. It transforms the original code used for characters of the text (ASCII code on 8 b, for instance). Coding the text is just replacing each symbol (more exactly, each occurrence of it) by its new codeword. The method works for any length of blocks (not only 8 b), but the running time grows exponentially with the length. In the following, we assume that symbols are originally encoded on 8 b to simplify the description.

The Huffman algorithm uses the notion of **prefix code**. A prefix code is a set of words containing no word that is a prefix of another word of the set. The advantage of such a code is that decoding is immediate. Moreover, it can be proved that this type of code does not weaken the compression.

A prefix code on the binary alphabet $\{0, 1\}$ can be represented by a trie (see section on the Aho–Corasick algorithm) that is a binary tree. In the present method codes are complete: they correspond to complete tries (internal nodes have exactly two children). The leaves are labeled by the original characters, edges are labeled by 0 or 1, and labels of branches are the words of the code. The condition on the code implies that codewords are identified with leaves only. We adopt the convention that, from an internal node, the edge to its left child is labeled by 0, and the edge to its right child is labeled by 1.

In the model where characters of the text are given new codewords, the Huffman algorithm builds a code that is optimal in the sense that the compression is the best possible (the length of the compressed text is minimum). The code depends on the text, and more precisely on the frequencies of each character in the uncompressed text. The more frequent characters are given short codewords, whereas the less frequent symbols have longer codewords.

13.7.1.1 Encoding

The coding algorithm is composed of three steps: count of character frequencies, construction of the prefix code, and encoding of the text.

The first step consists in counting the number of occurrences of each character in the original text (see Figure 13.52). We use a special end marker (denoted by END), which (virtually) appears only once at the end of the text. It is possible to skip this first step if fixed statistics on the alphabet are used. In this case, the method is optimal according to the statistics, but not necessarily for the specific text.

```
COUNT(fin)
for each character a ∈ Σ
    do freq(a) ← 0
while not end of file fin and a is the next symbol
    do freq(a) ← freq(a) + 1
freq(END) ← 1
```

FIGURE 13.52 Counts the character frequencies.

```
BUILD-TREE()
for each character a ∈ Σ ∪ {END}
    do if freq(a) ≠ 0
         then create a new node t
              weight(t) ← freq(a)
              label(t) ← a
lleaves ← list of all the nodes in increasing order of weight
ltrees ← empty list
while LENGTH(lleaves) + LENGTH(ltrees) > 1
    do (ℓ, r) ← extract the two nodes of smallest weight (among the two nodes at the
                  beginning of lleaves and the two nodes at the beginning of ltrees)
       create a new node t
       weight(t) ← weight(ℓ) + weight(r)
       left(t) ← ℓ
       right(t) ← r
       insert t at the end of ltrees
return t
```

FIGURE 13.53 Builds the coding tree.

The second step of the algorithm builds the tree of a prefix code using the character frequency $freq(a)$ of each character a in the following way:

- Create a one-node tree t for each character a, setting $weight(t) = freq(a)$ and $label(t) = a$,
- Repeat (1), extract the two least weighted trees t_1 and t_2, and (2) create a new tree t_3 having left subtree t_1, right subtree t_2, and weight $weight(t_3) = weight(t_1) + weight(t_2)$,
- Until only one tree remains.

The tree is constructed by the algorithm BUILD-TREE in Figure 13.53. The implementation uses two linear lists. The first list contains the leaves of the future tree, each associated with a symbol. The list is sorted in the increasing order of the weight of the leaves (frequency of symbols). The second list contains the newly created trees. Extracting the two least weighted trees consists in extracting the two least weighted trees among the two first trees of the list of leaves and the two first trees of the list of created trees. Each new tree is inserted at the end of the list of the trees. The only tree remaining at the end of the procedure is the coding tree.

After the coding tree is built, it is possible to recover the codewords associated with characters by a simple depth-first search of the tree (see Figure 13.54); $codeword(a)$ is then the binary code associated with the character a.

```
BUILD-CODE(t, length)
if t is not a leaf
  then temp[length] ← 0
       BUILD-CODE(left(t), length + 1)
       temp[length] ← 1
       BUILD-CODE(right(t), length + 1)
  else codeword(label(t)) ← temp[0 . . length − 1]
```

FIGURE 13.54 Builds the character codes from the coding tree.

```
CODE-TREE(fout, t)
if t is not a leaf
  then write a 0 in the file fout
       CODE-TREE(fout, left(t))
       CODE-TREE(fout, right(t))
  else write a 1 in the file fout
       write the original code of label(t) in the file fout
```

FIGURE 13.55 Memorizes the coding tree in the compressed file.

```
CODE-TEXT(fin, fout)
while not end of file fin and a is the next symbol
    do write codeword(a) in the file fout
write codeword(END) in the file fout
```

FIGURE 13.56 Encodes the characters in the compressed file.

```
CODING(fin, fout)
COUNT(fin)
t ← BUILD-TREE()
BUILD-CODE(t, 0)
CODE-TREE(fout, t)
CODE-TEXT(fin, fout)
```

FIGURE 13.57 Complete function for Huffman coding.

In the third step, the original text is encoded. Since the code depends on the original text, in order to be able to decode the compressed text, the coding tree and the original codewords of symbols must be stored with the compressed text. This information is placed in a header of the compressed file, to be read at decoding time just before the compressed text. The header is made via a depth-first traversal of the tree. Each time an internal node is encountered, a 0 is produced. When a leaf is encountered, a 1 is produced, followed by the original code of the corresponding character on 9 b (so that the end marker can be equal to 256 if all the characters appear in the original text). This part of the encoding algorithm is shown in Figure 13.55. After the header of the compressed file is computed, the encoding of the original text is realized by the algorithm of Figure 13.56.

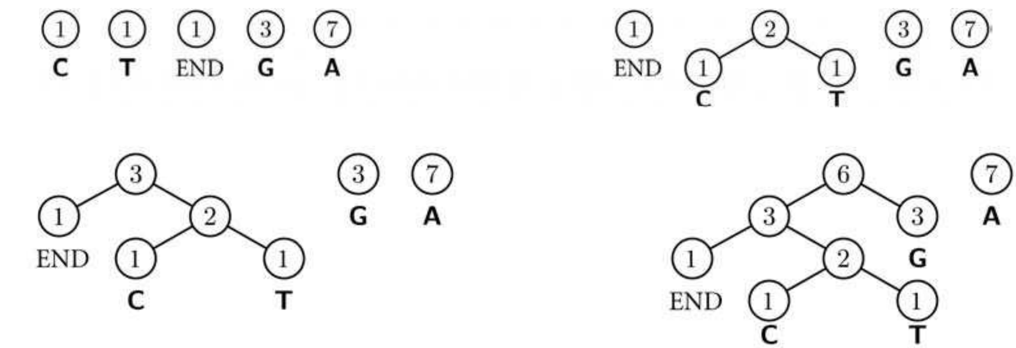
A complete implementation of the Huffman algorithm, composed of the three steps just described, is given in Figure 13.57.

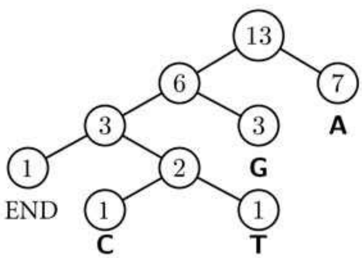
Example 13.21

Here, $y =$ **CAGATAAGAGAA**. The length of $y = 12 \times 8 = 96$ b (assuming an 8-b code). The character frequencies are

A	**C**	**G**	**T**	END
7	1	3	1	1

The different steps during the construction of the coding tree are





character codewords:

A	**C**	**G**	**T**	END
1	**0010**	**01**	**0011**	**000**

The encoded tree is **0001** binary (END, 9)**01**binary (**C**, 9)**1**binary(**T**, 9) **1**binary (**G**, 9)**1**binary (**A**, 9), which produces a header of length 54 b,

0001 100000000 01 001000011 1 001010100 1 001000111 1 001000001

The encoded text

0010 1 01 1 0011 1 1 01 1 01 1 1 000

is of length 24 b. The total length of the compressed file is 78 b.

The construction of the tree takes $O(\sigma \log \sigma)$ time if the sorting of the list of the leaves is implemented efficiently. The rest of the encoding process runs in linear time in the sum of the sizes of the original and compressed texts.

13.7.1.2 Decoding

Decoding a file containing a text compressed by the Huffman algorithm is a mere programming exercise. First, the coding tree is rebuilt by the algorithm of Figure 13.58. Then, the uncompressed text is recovered by parsing the compressed text with the coding tree. The process begins at the root of the coding tree and follows a left edge when a 0 is read or a right edge when a 1 is read. When a leaf is encountered, the corresponding character (in fact the original codeword of it) is produced and the parsing phase resumes at the root of the tree. The parsing ends when the codeword of the end marker is read. An implementation of the decoding of the text is presented in Figure 13.59.

```
REBUILD-TREE(fin, t)
b ← read a bit from the file fin
if b = 1                        ▷ leaf
  then left(t) ← NIL
       right(t) ← NIL
       label(t) ← symbol corresponding to the 9 next bits in the file fin
  else create a new node ℓ
       left(t) ← ℓ
       REBUILD-TREE(fin, ℓ)
       create a new node r
       right(t) ← r
       REBUILD-TREE(fin, r)
```

FIGURE 13.58 Rebuilds the tree read from the compressed file.

```
DECODE-TEXT(fin, fout, root)
t ← root
while label(t) ≠ END
    do if t is a leaf
          then label(t) in the file fout
               t ← root
          else b ← read a bit from the file fin
               if b = 1
                 then t ← right(t)
                 else t ← left(t)
```

FIGURE 13.59 Reads the compressed text and produces the uncompressed text.

```
DECODING(fin, fout)
create a new node root
REBUILD-TREE(fin, root)
DECODE-TEXT(fin, fout, root)
```

FIGURE 13.60 Complete function for Huffman decoding.

The complete decoding program is given in Figure 13.60. It calls the preceding functions. The running time of the decoding program is linear in the sum of the sizes of the texts it manipulates.

13.7.2 Lempel–Ziv–Welsh (LZW) Compression

Ziv and Lempel designed a compression method using encoding segments. These segments are stored in a dictionary that is built during the compression process. When a segment of the dictionary is encountered later while scanning the original text, it is substituted by its index in the dictionary. In the model where portions of the text are replaced by pointers on previous occurrences, the Ziv–Lempel compression scheme can be proved to be asymptotically optimal (on large enough texts satisfying good conditions on the probability distribution of symbols).

The dictionary is the central point of the algorithm. It has the property of being prefix closed (every prefix of a word of the dictionary is in the dictionary), so that it can be implemented as a tree. Furthermore, a hashing technique makes its implementation efficient. The version described in this section is called the Lempel–Ziv–Welsh method after several improvements introduced by Welsh. The algorithm is implemented by the **compress** command existing under the Unix operating system.

13.7.2.1 Compression Method

We describe the scheme of the compression method. The dictionary is initialized with all the characters of the alphabet. The current situation is when we have just read a segment w in the text. Let a be the next symbol (just following w). Then we proceed as follows:

- If wa is not in the dictionary, we write the index of w to the output file, and add wa to the dictionary. We then reset w to a and process the next symbol (following a).
- If wa is in the dictionary, we process the next symbol, with segment wa instead of w.

Initially, the segment w is set to the first symbol of the source text.

Example 13.22

Here $y =$ **CAGTAAGAGAA**

C	A	G	T	A	A	G	A	G	A	A		w	written	added
	↑											**C**	67	**CA**, 257
		↑										**A**	65	**AG**, 258
			↑									**G**	71	**GT**, 259
				↑								**T**	84	**TA**, 260
					↑							**A**	65	**AA**, 261
						↑						**A**		
							↑					**AG**	258	**AGA**, 262
								↑				**A**		
									↑			**AG**		
										↑		**AGA**	262	**AGAA**, 262
											↑	**A**		
													65	
													256	

13.7.2.2 Decompression Method

The decompression method is symmetrical to the compression algorithm. The dictionary is recovered while the decompression process runs. It is basically done in this way:

- Read a code c in the compressed file.
- Write in the output file the segment w that has index c in the dictionary.
- Add to the dictionary the word wa where a is the first letter of the next segment.

In this scheme, a problem occurs if the next segment is the word that is being built. This arises only if the text contains a segment $azazax$ for which az belongs to the dictionary but aza does not. During the compression process, the index of az is written into the compressed file, and aza is added to the dictionary. Next, aza is read and its index is written into the file. During the decompression process, the index of aza is read while the word az has not been completed yet: the segment aza is not already in the dictionary. However, because this is the unique case where the situation arises, the segment aza is recovered, taking the last segment az added to the dictionary concatenated with its first letter a.

Example 13.23

Here, the decoding is 67, 65, 71, 84, 65, 258, 262, 65, 256

read	written	added
67	**C**	
65	**A**	**CA**, 257
71	**G**	**AG**, 258
84	**T**	**GT**, 259
65	**A**	**TA**, 260
258	**AG**	**AA**, 261
262	**AGA**	**AGA**, 262
65	**A**	**AGAA**, 263
256		

13.7.2.3 Implementation

For the compression algorithm shown in Figure 13.61, the dictionary is stored in a table D. The dictionary is implemented as a tree; each node z of the tree has the three following components:

- *parent*(z) is a link to the parent node of z.
- *label*(z) is a character.
- *code*(z) is the code associated with z.

The tree is stored in a table that is accessed with a hashing function. This provides fast access to the children of a node. The procedure HASH-INSERT($(D, (p, a, c))$) inserts a new node z in the dictionary D with *parent*(z) $= p$, *label*(z) $= a$, and *code*(z) $= c$. The function HASH-SEARCH($(D, (p, a))$) returns the node z such that *parent*(z) $= p$ and *label*(z) $= a$.

```
COMPRESS(fin, fout)
count ← −1
for each character a ∈ Σ
    do count ← count + 1
        HASH-INSERT(D, (−1, a, count))
count ← count + 1
HASH-INSERT(D, (−1, END, count))
p ← −1
while not end of file fin
    do a ← next character of fin
        q ← HASH-SEARCH(D, (p, a))
        if q = NIL
          then write code(p) on 1 + log(count) bits in fout
                count ← count + 1
                HASH-INSERT(D, (p, a, count))
                p ← HASH-SEARCH(D, (−1, a))
          else  p ← q
write code(p) on 1 + log(count) bits in fout
write code(HASH-SEARCH(D, (−1, END))) on 1 + log(count) bits in fout
```

FIGURE 13.61 LZW compression algorithm.

```
UNCOMPRESS(fin, fout)
count ← −1
for each character a ∈ Σ
    do count ← count + 1
        HASH-INSERT(D, (−1, a, count))
count ← count + 1
HASH-INSERT(D, (−1, END, count))
c ← first code on 1 + log(count) bits in fin
write string(c) in fout
a ← first(string(c))
while TRUE
    do d ← next code on 1 + log(count) bits in fin
        if d > count
          then count ← count + 1
               parent(count) ← c
               label(count) ← a
               write string(c)a in fout
               c ← d
          else a ← first(string(d))
               if a ≠ END
                 then count ← count + 1
                      parent(count) ← c
                      label(count) ← a
                      write string(d) in fout
                      c ← d
                 else break
```

FIGURE 13.62 LZW decompression algorithm.

For the decompression algorithm, no hashing technique is necessary. Having the index of the next segment, a bottom-up walk in the trie implementing the dictionary produces the mirror image of the segment. A stack is used to reverse it. We assume that the function *string*(c) performs this specific work for a code c. The bottom-up walk follows the parent links of the data structure. The function *first*(w) gives the first character of the word w. These features are part of the decompression algorithm displayed in Figure 13.62.

The Ziv–Lempel compression and decompression algorithms run both in linear time in the sizes of the files provided a good hashing technique is chosen. Indeed, it is very fast in practice. Its main advantage compared to Huffman coding is that it captures long repeated segments in the source file.

13.7.3 Mixing Several Methods

We describe simple compression methods and then an example of a combination of several of them, the basis of the popular **bzip** software.

13.7.3.1 Run Length Encoding

The aim of Run Length Encoding (RLE) is to efficiently encode repetitions occurring in the input data. Let us assume that it contains a good quantity of repetitions of the form $aa \ldots a$ for some character a ($a \in \Sigma$). A repetition of k consecutive occurrences of letter a is replaced by &ak, where the symbol & is a new character (& $\notin \Sigma$).

The string $\&ak$ that encodes a repetition of k consecutive occurrences of a is itself encoded on the binary alphabet $\{0, 1\}$. In practice, letters are often represented by their ASCII code. Therefore, the codeword of a letter belongs to $\{0, 1\}^k$ with $k = 7$ or 8. Generally, there is no problem in choosing or encoding the special character &. The integer k of the string $\&ak$ is also encoded on the binary alphabet, but it is not sufficient to translate it by its binary representation, because we would be unable to recover it at decoding time inside the stream of bits. A simple way to cope with this is to encode k by the string $0^\ell \text{bin}(k)$, where $\text{bin}(k)$ is the binary representation of k, and ℓ is the length. This works well because the binary representation of k starts with a 1 so there is no ambiguity to recover ℓ by counting during the decoding phase. The size of the encoding of k is thus roughly $2 \log k$. More sophisticated integer representations are possible, but none is really suitable for the present situation. Simpler solution consists in encoding k on the same number of bits as other symbols, but this bounds values of ℓ and decreases the power of the method.

13.7.3.2 Move To Front

The Move To Front (MTF) method can be regarded as an extension of Run Length Encoding or a simplification of Ziv–Lempel compression. It is efficient when the occurrences of letters in the input text are localized into a relatively short segment of it. The technique is able to capture the proximity between occurrences of symbols and to turn it into a short encoded text.

Letters of the alphabet Σ of the input text are initially stored in a list that is managed dynamically. Letters are represented by their rank in the list, starting from 1, rank that is itself encoded as described above for RLE.

Letters of the input text are processed in an on-line manner. The clue of the method is that each letter is moved to the beginning of the list just after it is translated by the encoding of its rank.

The effect of MTF is to reduce the size of the encoding of a letter that reappears soon after its preceding occurrence.

13.7.3.3 Integrated Example

Most compression software combines several methods to be able to efficiently compress a large range of input data. We present an example of this strategy, implemented by the UNIX command **bzip**.

Let $y = y[0]y[1] \cdots y[n-1]$ be the input text. The k-th rotation (or conjugate) of y, $0 \le k \le n-1$, is the string $y_k = y[k]y[k+1] \cdots y[n-1]y[0]y[1] \cdots y[k-1]$.

We define the *BW* transformation as $BW(y) = y[p_0]y[p_1] \cdots y[p_{n-1}]$, where $p_i + 1$ is such that y_{p_i+1} has rank i in the sorted list of all rotations of y.

It is remarkable that y can be recovered from both $BW(y)$ and a position on it, starting position of the inverse transformation (see Figure 13.63). This is possible due to the following property of the transformation. Assume that $i < j$ and $y[p_i] = y[p_j] = a$. Since $i < j$, the definition implies $y_{p_i+1} < y_{p_j+1}$. Since $y[p_i] = y[p_j]$, transferring the last letters of y_{p_i+1} and y_{p_j+1} to the beginning of these words does not change the inequality. This proves that the two occurrences of a in $BW(y)$ are in the same relative order as in the sorted list of letters of y. Figure 13.63 illustrates the inverse transformation.

Transformation *BW* obviously does not compress the input text y. But $BW(y)$ is compressed more efficiently with simple methods. This is the strategy applied for the command **bzip**. It is a combination of the *BW* transformation followed by MTF encoding and RLE encoding. Arithmetic coding, a method providing compression ratios slightly better than Huffman coding, can also be used.

Table 13.1 contains a sample of experimental results showing the behavior of compression algorithms on different types of texts from the Calgary Corpus: `bib` (bibliography), `book1` (fiction book), `news` (USENET batch file), `pic` (black and white fax picture), `progc` (source code in C), and `trans` (transcript of terminal session).

The compression algorithms reported in the table are the Huffman coding algorithm implemented by **pack**, the Ziv–Lempel algorithm implemented by **gzip-b**, and the compression based on the *BW* transform implemented by **bzip2-1**.

Additional compression results can be found at `http://corpus.canterbury.ac.nz`.

TABLE 13.1 Compression Results with Three Algorithms. Huffman coding (**pack**), Ziv–Lempel coding (**gzip-b**) and Burrows-Wheeler coding (**bzip2-1**). Figures give the number of bits used per character (letter). They show that **pack** is the less efficient method and that **bzip2-1** compresses a bit more than **gzip-b**.

Sizes in bytes	111,261	768,771	377,109	513,216	39,611	93,695	
Source Texts	`bib`	`book1`	`news`	`pic`	`progc`	`trans`	Average
pack	5.24	4.56	5.23	1.66	5.26	5.58	4.99
gzip-b	2.51	3.25	3.06	0.82	2.68	1.61	2.69
bzip2-1	2.10	2.81	2.85	0.78	2.53	1.53	2.46

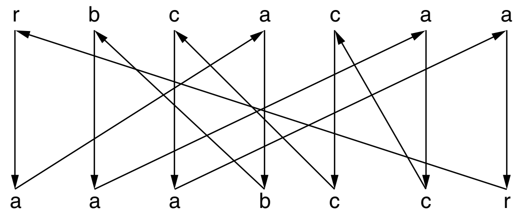


FIGURE 13.63 Example of text y = **baccara**. Top line is $BW(y)$ and bottom line the sorted list of letters of it. Top-down arrows correspond to succession of occurrences in y. Each bottom-up arrow links the same occurrence of a letter in y. Arrows starting from equal letters do not cross. The circular path is associated with rotations of the string y. If the starting point is known, the only occurrence of letter **b** here, following the path produces the initial string y.

13.8 Research Issues and Summary

The algorithm for string searching by hashing was introduced by Harrison in 1971, and later fully analyzed by Karp and Rabin (1987).

The linear-time string-matching algorithm of Knuth, Morris, and Pratt is from 1976. It can be proved that, during the search, a character of the text is compared to a character of the pattern no more than $\log_\Phi(|x| + 1)$ (where Φ is the golden ratio $(1 + \sqrt{5})/2$). Simon (1993) gives an algorithm similar to the previous one but with a delay bounded by the size of the alphabet (of the pattern x). Hancart (1993) proves that the delay of Simon's algorithm is, indeed, no more than $1 + \log_2 |x|$. He also proves that this is optimal among algorithms searching the text through a window of size 1.

Galil (1981) gives a general criterion to transform searching algorithms of that type into real-time algorithms.

The Boyer–Moore algorithm was designed by Boyer and Moore (1977). The first proof on the linearity of the algorithm when restricted to the search of the first occurrence of the pattern is in Knuth et al. (1977). Cole (1994) proves that the maximum number of symbol comparisons is bounded by $3n$, and that this bound is tight.

Knuth et al. (1977) consider a variant of the Boyer–Moore algorithm in which all previous matches inside the current window are memorized. Each window configuration becomes the state of what is called the Boyer–Moore automaton. It is still unknown whether the maximum number of states of the automaton is polynomial or not.

Several variants of the Boyer–Moore algorithm avoid the quadratic behavior when searching for all occurrences of the pattern. Among the more efficient in terms of the number of symbol comparisons are the algorithm of Apostolico and Giancarlo (1986), Turbo–BM algorithm by Crochemore et al. (1992) (the two algorithms are analyzed in Lecroq (1995)), and the algorithm of Colussi (1994).

The general bound on the expected time complexity of string matching is $O(|y| \log |x|/|x|)$. The probabilistic analysis of a simplified version of the Boyer–Moore algorithm, similar to the Quick Search algorithm of Sunday (1990) described in the chapter, was studied by several authors.

String searching can be solved by a linear-time algorithm requiring only a constant amount of memory in addition to the pattern and the (window on the) text. This can be proved by different techniques presented in Crochemore and Rytter (2002).

The Aho–Corasick algorithm is from Aho and Corasick (1975). It is implemented by the **fgrep** command under the UNIX operating system. Commentz-Walter (1979) has designed an extension of the Boyer-Moore algorithm to several patterns. It is fully described in Aho (1990).

On general alphabets the two-dimensional pattern matching can be solved in linear time, whereas the running time of the Bird/Baker algorithm has an additional $\log \sigma$ factor. It is still unknown whether the problem can be solved by an algorithm working simultaneously in linear time and using only a constant amount of memory space (see Crochemore and Rytter 2002).

The suffix tree construction of Section 13.2 is by McCreight (1976). An on-line construction is given by Ukkonen (1995). Other data structures to represent indexes on text files are: direct acyclic word graph (Blumer et al., 1985), suffix automata (Crochemore, 1986), and suffix arrays (Manber and Myers, 1993). All these techniques are presented in (Crochemore and Rytter, 2002). The data structures implement full indexes with standard operations, whereas applications sometimes need only incomplete indexes. The design of compact indexes is still unsolved.

First algorithms for aligning two sequences are by Needleman and Wunsch (1970) and Wagner and Fischer (1974). Idea and algorithm for local alignment is by Smith and Waterman (1981). Hirschberg (1975) presents the computation of the lcs in linear space. This is an important result because the algorithm is classically run on large sequences. Another implementation is given in Durbin et al. (1998). The quadratic time complexity of the algorithm to compute the Levenshtein distance is a bottleneck in practical string comparison for the same reason.

Approximate string searching is a lively domain of research. It includes, for instance, the notion of regular expressions to represent sets of strings. Algorithms based on regular expression are commonly found in books related to compiling techniques. The algorithms of Section 13.6 are by Baeza-Yates and Gonnet (1992) and Wu and Manber (1992).

The statistical compression algorithm of Huffman (1951) has a dynamic version where symbol counting is done at coding time. The current coding tree is used to encode the next character and then updated. At decoding time, a symmetrical process reconstructs the same tree, so the tree does not need to be stored with the compressed text; see Knuth (1985). The command **compact** of UNIX implements this version.

Several variants of the Ziv and Lempel algorithm exist. The reader can refer to Bell et al. (1990) for further discussion. Nelson (1992) presents practical implementations of various compression algorithms. The *BW* transform is from Burrows and Wheeler (1994).

Defining Terms

Alignment: An alignment of two strings x and y is a word of the form $(\overline{x}_0, \overline{y}_0)(\overline{x}_1, \overline{y}_1)\cdots(\overline{x}_{p-1}, \overline{y}_{p-1})$ where each $(\overline{x}_i, \overline{y}_i) \in (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus (\{(\varepsilon, \varepsilon)\}$ for $0 \leq i \leq p-1$ and both $x = \overline{x}_0\overline{x}_1\cdots\overline{x}_{p-1}$ and $y = \overline{y}_0\overline{y}_1\cdots\overline{y}_{p-1}$.

Border: A word $u \in \Sigma^*$ is a border of a word $w \in \Sigma^*$ if u is both a prefix and a suffix of w (there exist two words $v, z \in \Sigma^*$ such that $w = vu = uz$). The common length of v and z is a period of w.

Edit distance: The metric distance between two strings that counts the minimum number of insertions and deletions of symbols to transform one string into the other.

Hamming distance: The metric distance between two strings of same length that counts the number of mismatches.

Levenshtein distance: The metric distance between two strings that counts the minimum number of insertions, deletions, and substitutions of symbols to transform one string into the other.

Occurrence: An occurrence of a word $u \in \Sigma^*$, of length m, appears in a word $w \in \Sigma^*$, of length n, at position i if for $0 \leq k \leq m-1$, $u[k] = w[i+k]$.

Prefix: A word $u \in \Sigma^*$ is a prefix of a word $w \in \Sigma^*$ if $w = uz$ for some $z \in \Sigma^*$.

Prefix code: Set of words such that no word of the set is a prefix of another word contained in the set. A prefix code is represented by a coding tree.

Segment: A word $u \in \Sigma^*$ is a segment of a word $w \in \Sigma^*$ if u occurs in w (see occurrence); that is, $w = vuz$ for two words $v, z \in \Sigma^*$ (u is also referred to as a factor or a subword of w).

Subsequence: A word $u \in \Sigma^*$ is a subsequence of a word $w \in \Sigma^*$ if it is obtained from w by deleting zero or more symbols that need not be consecutive (u is sometimes referred to as a subword of w, with a possible confusion with the notion of segment).

Suffix: A word $u \in \Sigma^*$ is a suffix of a word $w \in \Sigma^*$ if $w = vu$ for some $v \in \Sigma^*$.

Suffix tree: Trie containing all the suffixes of a word.

Trie: Tree in which edges are labeled by letters or words.

References

Aho, A.V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*, Vol. A. *Algorithms and Complexity*, J. van Leeuwen, Ed., pp. 255–300. Elsevier, Amsterdam.

Aho, A.V. and Corasick, M.J. 1975. Efficient string matching: an aid to bibliographic search. *Comm. ACM*, 18(6):333–340.

Baeza-Yates, R.A. and Gonnet, G.H. 1992. A new approach to text searching. *Comm. ACM*, 35(10):74–82.

Baker, T.P. 1978. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.*, 7(4):533–541.

Bell, T.C., Cleary, J.G., and Witten, I.H. 1990. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.

Bird, R.S. 1977. Two-dimensional pattern matching. *Inf. Process. Lett.*, 6(5):168–170.

Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T., and Seiferas, J. 1985. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55.

Boyer, R.S. and Moore, J.S. 1977. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772.

Breslauer, D., Colussi, L., and Toniolo, L. 1993. Tight comparison bounds for the string prefix matching problem. *Inf. Process. Lett.*, 47(1):51–57.

Burrows, M. and Wheeler, D. 1994. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.

Cole, R. 1994. Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM J. Comput.*, 23(5):1075–1091.

Colussi, L. 1994. Fastest pattern matching in strings. *J. Algorithms*, 16(2):163–189.

Crochemore, M. 1986. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86.

Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology*. World Scientific.

Durbin, R., Eddy, S., and Krogh, A., and Mitchison G. 1998. *Biological Sequence Analysis Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

Galil, Z. 1981. String matching in real time. *J. ACM*, 28(1):134–149.

Hancart, C. 1993. On Simon's string searching algorithm. *Inf. Process. Lett.*, 47(2):95–99.

Hirschberg, D.S. 1975. A linear space algorithm for computing maximal common subsequences. *Comm. ACM*, 18(6):341–343.

Hume, A. and Sunday, D.M. 1991. Fast string searching. *Software — Practice Exp.*, 21(11):1221–1248.

Karp, R.M. and Rabin, M.O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260.

Knuth, D.E. 1985. Dynamic Huffman coding. *J. Algorithms*, 6(2):163–180.

Knuth, D.E., Morris, J.H., Jr, and Pratt, V.R. 1977. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350.

Lecroq, T. 1995. Experimental results on string-matching algorithms. *Software — Practice Exp.* 25(7): 727–765.

McCreight, E.M. 1976. A space-economical suffix tree construction algorithm. *J. Algorithms*, 23(2): 262–272.

Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948.
Needleman, S.B. and Wunsch, C.D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453.
Nelson, M. 1992. *The Data Compression Book*. M&T Books.
Simon, I. 1993. String matching algorithms and automata. In *First American Workshop on String Processing*, Baeza-Yates and Ziviani, Eds., pp. 151–157. Universidade Federal de Minas Gerais.
Smith, T.F. and Waterman, M.S. 1981. Identification of common molecular sequences. *J. Mol. Biol.*, 147:195–197.
Stephen, G.A. 1994. *String Searching Algorithms*. World Scientific Press.
Sunday, D.M. 1990. A very fast substring search algorithm. *Commun. ACM* 33(8):132–142.
Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.
Wagner, R.A. and Fischer, M. 1974. The string-to-string correction problem. *J. ACM*, 21(1):168–173.
Welch, T. 1984. A technique for high-performance data compression. *IEEE Comput.* 17(6):8–19.
Wu, S. and Manber, U. 1992. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91.
Zhu, R.F. and Takaoka, T. 1989. A technique for two-dimensional pattern matching. *Commun. ACM*, 32(9):1110–1120.

Further Information

Problems and algorithms presented in the chapter are just a sample of questions related to pattern matching. They share the formal methods used to design solutions and efficient algorithms. A wider panorama of algorithms on texts can be found in books, other including:

Apostolico, A. and Galil, Z., Editors. 1997. *Pattern Matching Algorithms*. Oxford University Press.
Bell, T.C., Cleary, J.G., and Witten, I.H. 1990. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.
Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology.* World Scientific.
Gusfield D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.
Navarro, G. and Raffinot M. 2002. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press.
Nelson, M. 1992. *The Data Compression Book.* M&T Books.
Salomon, D. 2000. *Data Compression: the Complete Reference*. Springer-Verlag.
Stephen, G.A. 1994. *String Searching Algorithms.* World Scientific Press.

Research papers in pattern matching are disseminated in a few journals, among which are: *Communications of the ACM*, *Journal of the ACM*, *Theoretical Computer Science*, *Algorithmica*, *Journal of Algorithms*, *SIAM Journal on Computing*, and *Journal of Discrete Algorithms.*

Finally, three main annual conferences present the latest advances of this field of research and Combinatorial Pattern Matching, which started in 1990. Data Compression Conference, which is regularly held at Snowbird. The scope of SPIRE (String Processing and Information Retrieval) includes the domain of data retrieval.

General conferences in computer science often have sessions devoted to pattern matching algorithms.

Several books on the design and analysis of general algorithms contain chapters devoted to algorithms on texts. Here is a sample of these books:

Cormen, T.H., Leiserson, C.E., and Rivest, R.L. 1990. *Introduction to Algorithms*. MIT Press.
Gonnet, G.H. and Baeza-Yates, R.A. 1991. *Handbook of Algorithms and Data Structures*. Addison-Wesley.

Animations of selected algorithms can be found at:

`http://www-igm.univ-mlv.fr/~lecroq/string/` (Exact String Matching Algorithms),
`http://www-igm.univ-mlv.fr/~lecroq/seqcomp/` (Alignments).

14

Genetic Algorithms

Stephanie Forrest
University of New Mexico

14.1 Introduction

A genetic algorithm is a form of evolution that occurs in a computer. Genetic algorithms are useful, both as search methods for solving problems and for modeling evolutionary systems. This chapter describes how genetic algorithms work, gives several examples of genetic algorithm applications, and reviews some mathematical analysis of genetic algorithm behavior.

In genetic algorithms, strings of binary digits are stored in a computer's memory, and over time the properties of these strings evolve in much the same way that populations of individuals evolve under natural selection. Although the computational setting is highly simplified when compared with the natural world, genetic algorithms are capable of evolving surprisingly complex and interesting structures. These structures, called **individuals**, can represent solutions to problems, strategies for playing games, visual images, or computer programs. Thus, genetic algorithms allow engineers to use a computer to evolve problem solutions over time, instead of designing them by hand. Although genetic algorithms are known primarily as a problem-solving method, they can also be used to study and model evolution in various settings, including biological (such as ecologies, immunology, and population genetics), social (such as economies and political systems), and cognitive systems.

14.2 Underlying Principles

The basic idea of a genetic algorithm is quite simple. First, a population of individuals is created in a computer, and then the population is evolved using the principles of variation, selection, and inheritance. Random variations in the population result in some individuals being more fit than others (better suited to their environment). These individuals have more offspring, passing on successful variations to their children, and the cycle is repeated. Over time, the individuals in the population become better adapted to their environment. There are many ways of implementing this simple idea. Here I describe the one invented by Holland [1975, Goldberg 1989].

The idea of using selection and variation to evolve solutions to problems goes back at least to Box [1957], although his work did not use a computer. In the late 1950s and early 1960s there were several independent

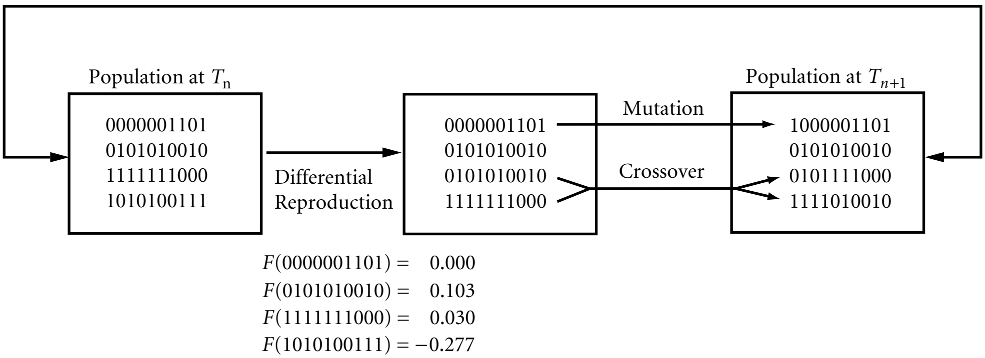


FIGURE 14.1 (See Plate 14.1 in the color insert following page **29**-22.) Genetic algorithm overview: A population of four individuals is shown. Each is assigned a fitness value by the function $F(x, y) = yx^2 - x^4$. (See Figure 14.3.) On the basis of these fitnesses, the selection phase assigns the first individual (0000001101) one copy, the second (0101010010) two copies, the third (1111111000) one copy, and the fourth (1010100111) zero copies. After selection, the genetic operators are applied probabilistically; the first individual has its first bit mutated from a 0 to a 1, and crossover combines the last two individuals into two new ones. The resulting population is shown in the box labeled $T_{(N+1)}$.

efforts to incorporate ideas from evolution in computation. Of these, the best known are genetic algorithms [Holland 1962], evolutionary programming [Fogel et al. 1966], and evolutionary strategies [Back and Schwefel 1993]. Rechenberg [Back and Schwefel 1993] emphasized the importance of selection and mutation as mechanisms for solving difficult real-valued optimization problems. Fogel et al. [1966] developed similar ideas for evolving intelligent agents in the form of finite state machines. Holland [1962, 1975] emphasized the adaptive properties of entire populations and the importance of recombination mechanisms such as **crossover**. In recent years, genetic algorithms have taken many forms, and in some cases bear little resemblance to Holland's original formulation. Researchers have experimented with different types of representations, crossover and mutation operators, special-purpose operators, and different approaches to reproduction and selection. However, all of these methods have a family resemblance in that they take some inspiration from biological evolution and from Holland's original genetic algorithm. A new term, *evolutionary computation*, has been introduced to cover these various members of the genetic algorithm family, evolutionary programming, and evolution strategies.

Figure 14.1 gives an overview of a simple genetic algorithm. In its simplest form, each individual in the population is a bit string. Genetic algorithms often use more complex representations, including richer alphabets, diploidy, redundant encodings, and multiple **chromosomes**. However, the binary case is both the simplest and the most general. By analogy with genetics, the string of bits is referred to as the **genotype**. Each individual consists only of its genetic material, and it is organized into one (haploid) chromosome. Each bit position (set to 1 or 0) represents one gene. I will use the term bit string to refer both to genotypes and the individuals that they define. A natural question is how genotypes built from simple strings of bits can specify a solution to a specific problem. In other words, how are the binary genes expressed? There are many techniques for mapping bit strings to different problem domains, some of which are described in the following subsections.

The initial population of individuals is usually generated randomly, although it need not be. For example, prior knowledge about the problem solution can be encoded directly into the initial population, as in Hillis [1990]. Each individual is tested empirically in an environment, receiving a numerical evaluation of its merit, assigned by a **fitness function** F. The environment can be almost anything: another computer simulation, interactions with other individuals in the population, actions in the physical world (by a robot for example), or a human's subjective judgment. The fitness function's evaluation typically returns a single number (usually, higher numbers are assigned to fitter individuals). This constraint is sometimes relaxed so that the fitness function returns a vector of numbers [Fonseca and Fleming 1995], which can be appropriate for problems with multiple objectives. The fitness function determines how each gene (bit)

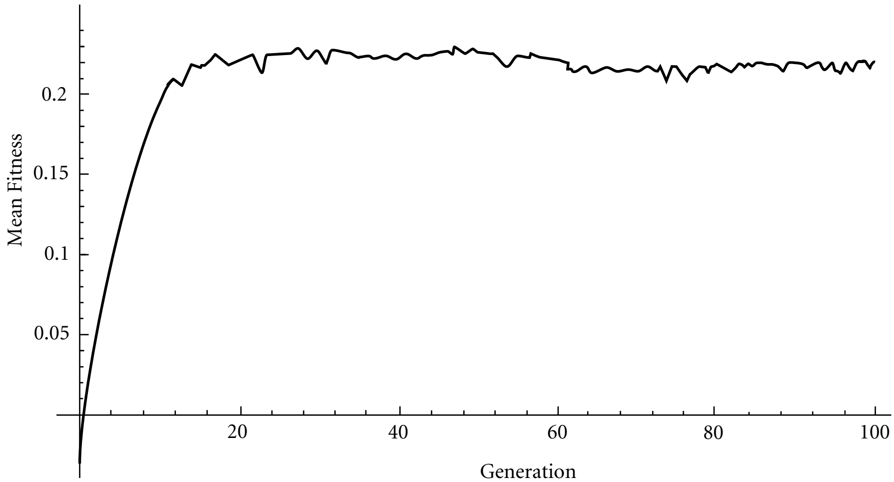


FIGURE 14.2 Mean fitness of a population evolving under the genetic algorithm. The population size is 100 individuals, each of which is 10 bits long (5 bits for x, 5 bits for y, as described in Figure 14.3), mutation probability is 0.0026/bit, crossover probability is 0.6 per pair of individuals, and the fitness function is $F = yx^2 - x^4$. Population mean is shown every generation for 100 generations.

of an individual will be interpreted and thus what specific problem the population will evolve to solve. The fitness function is the primary place where the traditional genetic algorithm is tailored to a specific problem.

Once all individuals in the population have been evaluated, their fitnesses form the basis for selection. **Selection** is implemented by eliminating low-fitness individuals from the population, and inheritance is implemented by making multiple copies of high-fitness individuals. Genetic operators such as **mutation** (flipping individual bits) and crossover (exchanging substrings of two individuals to obtain new offspring) are then applied probabilistically to the selected individuals to produce a new population (or **generation**) of individuals. The term crossover is used here to refer to the exchange of homologous substrings between individuals, although the biological term crossing over generally implies exchange within an individual. New generations can be produced either synchronously, so that the old generation is completely replaced, or asynchronously, so that generations overlap.

By transforming the previous set of good individuals to a new one, the operators generate a new set of individuals that ideally have a better than average chance of also being good. When this cycle of evaluation, selection, and genetic operations is iterated for many generations, the overall fitness of the population generally improves, as shown in Figure 14.2, and the individuals in the population represent improved solutions to whatever problem was posed in the fitness function.

There are many details left unspecified by this description. For example, selection can be performed in any of several ways — it could arbitrarily eliminate the least fit 50% of the population and make one copy of all of the remaining individuals, it could replicate individuals in direct proportion to their fitness (fitness-proportionate selection), or it could scale the fitnesses in any of several ways and replicate individuals in direct proportion to their scaled values (a more typical method). Similarly, the crossover operator can pass on both offspring to the new generation, or it can arbitrarily choose one to be passed on; the number of crossover points can be restricted to one per pair, two per pair, or N per pair. These and other variations of the basic algorithm have been discussed extensively in Goldberg [1989], in Davis [1991], and in the Proceedings of the International Conference on Genetic Algorithms. (See Further Information section.)

The genetic algorithm is interesting from a computational standpoint, at least in part, because of the claims that have been made about its effectiveness as a biased sampling algorithm. The classical argument

about genetic algorithm performance has three components [Holland 1975, Goldberg 1989]:

- Independent sampling is provided by large populations that are initialized randomly.
- High-fitness individuals are preserved through selection, and this biases the sampling process toward regions of high fitness.
- Crossover combines partial solutions, called building blocks, from different strings onto the same string, thus exploiting the parallelism provided by the population of candidate solutions.

A partial solution is taken to be a hyperplane in the search space of strings and is called a **schema** (see Section 14.4). A central claim about genetic algorithms is that schemas capture important regularities in the search space and that a form of *implicit parallelism* exists because one fitness evaluation of an individual comprising l bits implicitly gives information about the 2^l schemas, or hyperplanes, of which it is an instance. The Schema Theorem states that the genetic algorithm operations of reproduction, mutation, and crossover guarantee exponentially increasing samples of the observed best schemas in the next time step. By analogy with the k-armed bandit problem it can be argued that the genetic algorithm uses an optimal sampling strategy [Holland 1975]. See Section 14.4 for details.

14.3 Best Practices

The simple computational procedure just described can be applied in many different ways to solve a wide range of problems. In designing a genetic algorithm to solve a specific problem there are two major design decisions: (1) specifying the mapping between binary strings and candidate solutions (this is commonly referred to as the representation problem) and (2) defining a concrete measure of fitness. In some cases the best representation and fitness function are obvious, but in many cases they are not, and in all cases, the particular representation and fitness function that are selected will determine the ultimate success of the genetic algorithm on the chosen problem. Possibly the simplest representation is a *feature list* in which each bit, or gene, represents the presence or absence of a single feature. This representation is useful for learning pattern classes defined by a critical set of features. For example, in spectroscopic applications, an important problem is selecting a small number of spectral frequencies that predict the concentration of some substance (e.g., concentration of glucose in human blood). The feature list approach to this problem assigns 1 bit to represent the presence or absence of each different observable frequency, and high fitness is assigned to those individuals whose feature settings correspond to good predictors for high (or low) glucose levels [Thomas 1993].

Genetic algorithms in various forms have been applied to many scientific and engineering problems, including optimization, automatic programming, machine and robot learning, modeling natural systems, and artificial life. They have been used in a wide variety of optimization tasks, including numerical optimization (see section on function optimization) and combinatorial optimization problems such as circuit design and job shop scheduling (see section on ordering problems). Genetic algorithms have also been used to evolve computer programs for specific tasks (see section on automatic programming) and to design other computational structures, e.g., cellular automata rules and sorting networks. In machine learning, they have been used to design neural networks, to evolve rules for rule-based systems, and to design and control robots. For an overview of genetic algorithms in machine learning, see DeJong [1990a, 1990b] and Schaffer et al. [1992].

Genetic algorithms have been used to model processes of innovation, the development of bidding strategies, the emergence of economic markets, the natural immune system, and ecological phenomena such as biological arms races, host–parasite coevolution, symbiosis, and resource flow. They have been used to study evolutionary aspects of social systems, such as the evolution of cooperation, the evolution of communication, and trail-following behavior in ants. They have been used to study questions in population genetics, such as “under what conditions will a gene for recombination be evolutionarily viable?” Finally, genetic algorithms are an important component in many artificial-life models, including systems that model interactions between species evolution and individual learning. See Further Information section and Mitchell and Forrest [1994] for details about genetic algorithms in modeling and artificial life.

The remainder of this section describes four illustrative examples of how genetic algorithms are used: numerical encodings for function optimization, permutation representations and special operators for sequencing problems, computer programs for automated programming, and endogenous fitness and other extensions for ecological modeling. The first two cover the most common classes of engineering applications. They are well understood and noncontroversial. The third example illustrates one of the most promising recent advances in genetic algorithms, but it was developed more recently and is less mature than the first two. The final example shows how genetic algorithms can be modified to more closely approximate natural evolutionary processes.

14.3.1 Function Optimization

Perhaps the most common application of genetic algorithms, pioneered by DeJong [1975], is multiparameter function optimization. Many problems can be formulated as a search for an optimal value, where the value is a complicated function of some input parameters. In some cases, the parameter settings that lead to the exact greatest (or least) value of the function are of interest. In other cases, the exact optimum is not required, just a near optimum, or even a value that represents a slight improvement over the previously best-known value. In these latter cases, genetic algorithms are often an appropriate method for finding good values.

As a simple example, consider the function $f(x, y) = yx^2 - x^4$. This function is solvable analytically, but if it were not, a genetic algorithm could be used to search for values of x and y that produce high values of $f(x, y)$ in a particular region of $\Re^2$. The most straightforward representation (Figure 14.3) is to assign regions of the bit string to represent each parameter (variable). Once the order in which the parameters are to appear is determined (in the figure x appears first and y appears second), the next step is to specify the domain for x and y (that is, the set of values for x and y that are candidate solutions). In our example, x and y will be real values in the interval $[0, 1)$. Because x and y are real valued in this example, and we are using a bit representation, the parameters need to be discretized. The precision of the solution is determined by how many bits are used to represent each parameter. In the example, 5 bits are assigned for x and 5 for y, although 10 is a more typical number. There are different ways of mapping between bits and decimal numbers, and so an encoding must also be chosen, and here we use gray coding.

Once a representation has been chosen, the genetic algorithm generates a random population of bit strings, decodes each bit string into the corresponding decimal values for x and y, applies the fitness function ($f(x, y) = yx^2 - x^4$) to the decoded values, selects the most fit individuals [those with the highest $f(x, y)$] for copying and variation, and then repeats the process. The population will tend to converge on a set of bit strings that represents an optimal or near optimal solution. However, there will always be some variation in the population due to mutation (Figure 14.2).

The standard binary encoding of decimal values has the drawback that in some cases all of the bits must be changed in order to increase a number by one. For example, the bit pattern 011 translates to 3 in decimal,

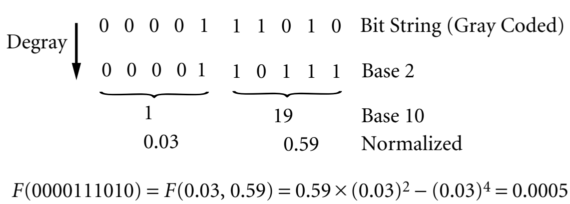


FIGURE 14.3 Bit-string encoding of multiple real-valued parameters. An arbitrary string of 10 bits is interpreted in the following steps: (1) segment the string into two regions with the first 5 bits reserved for x and the second 5 bits for y; (2) interpret each 5-bit substring as a Gray code and map back to the corresponding binary code; (3) map each 5-bit substring to its decimal equivalent; (4) scale to the interval $[0, 1)$; (5) substitute the two scaled values for x and y in the fitness function F; (6) return $F(x, y)$ as the fitness of the original string.

but 4 is represented by 100. This can make it difficult for an individual that is close to an optimum to move even closer by mutation. Also, mutations in high-order bits (the leftmost bits) are more significant than mutations in low-order bits. This can violate the idea that bit strings in successive generations will have a better than average chance of having high fitness, because mutations may often be disruptive. Gray codes address the first of these problems. Gray codes have the property that incrementing or decrementing any number by one is always 1 bit change. In practice, Gray-coded representations are often more successful for multiparameter function optimization applications of genetic algorithms.

Many genetic algorithm practitioners encode real-valued parameters directly without converting to a bit-based representation. In this approach, each parameter can be thought of as a gene on the chromosome. Crossover is defined as before, except that crosses take place only between genes (between real numbers). Mutation is typically redefined so that it chooses a random value that is close to the current value. This representation strategy is often more effective in practice, but it requires some modification of the operators [Back and Schwefel 1993, Davis 1991]. There are a number of other representation tricks that are commonly employed for function optimization, including logarithmic scaling (interpreting bit strings as the logarithm of the true parameter value), dynamic encoding (a technique that allows the number and interpretation of bits allocated to a particular parameter to vary throughout a run), variable-length representations, delta coding (the bit strings express a distance away from some previous partial solution), and a multitude of nonbinary encodings.

This completes our description of a simple method for encoding parameters onto a bit string. Although a function of two variables was used as an example, the strength of the genetic algorithm lies in its ability to manipulate many parameters, and this method has been used for hundreds of applications, including aircraft design, tuning parameters for algorithms that detect and track multiple signals in an image, and locating regions of stability in systems of nonlinear difference equations. See Goldberg [1989], Davis [1991], and the Proceedings of the International Conference on Genetic Algorithms for more detail about these and other examples of successful function-optimization applications.

14.3.2 Ordering Problems

A common problem involves finding an optimal ordering for a sequence of *N* items. Examples include various NP-complete problems such as finding a tour of cities that minimizes the distance traveled (the traveling salesman problem), packing boxes into a bin to minimize wasted space (the bin packing problem), and graph coloring problems.

For example, in the traveling salesman problem, suppose there are four cities: 1, 2, 3, and 4 and that each city is labeled by a unique bit string.* A common fitness function for this problem is the length of the candidate tour. A natural way to represent a tour is as a permutation, so that 3 2 1 4 is one candidate tour and 4 1 2 3 is another. This representation is problematic for the genetic algorithm because mutation and crossover do not necessarily produce legal tours. For example, a crossover between positions two and three in the example produces the individuals 3 2 2 3 and 4 1 1 4, both of which are illegal tours — not all of the cities are visited and some are visited more than once.

Three general methods have been proposed to address this representation problem: (1) adopting a different representation, (2) designing specialized crossover operators that produce only legal tours, and (3) penalizing illegal solutions through the fitness function. Of these, the use of specialized operators has been the most successful method for applications of genetic algorithms to ordering problems such as the traveling salesman problem (for example, see Mühlenbein et al. [1988]), although a number of generic representations have been proposed and used successfully on other sequencing problems. Specialized crossover operators tend to be less general, and I will describe one such method, **edge recombination**, as an example of a special-purpose operator that can be used with the permutation representation already described.

*For simplicity, we will use integers in the following explanation rather than the bit strings to which they correspond.

Original Individuals		New Individual
3 6 2 1 4 5	$\longrightarrow$	3 6 4 1 2 5
5 2 1 3 6 4		

Adjacency List

Key	Adjacent Keys
1	2, 2, 3, 4
2	1, 1, 3, 6
3	1, 6, 6
4	1, 5, 6
5	2, 4
6	2, 3, 3, 4

FIGURE 14.4 Example of edge-recombination operator. The adjacency list is constructed by examining each element in the parent permutations (labeled Key) and recording its adjacent elements. The new individual is constructed by selecting one parent arbitrarily (the top parent) and assigning its first element (3) to be the first element in the new permutation. The adjacencies of 3 are examined, and 6 is chosen to be the second element because it is a shared adjacency. The adjacencies of 6 are then examined, and of the unused ones, 4 is chosen randomly. Similarly, 1 is assigned to be the fourth element in the new permutation by random choice from $\{1, 5\}$. Then 2 is placed as the fifth element because it is a shared adjacency, and then the one remaining element, 5, is placed in the last position.

When designing special-purpose operators it is important to consider what information from the parents is being transmitted to the offspring, that is, what information is correlated with high-fitness individuals. In the case of traditional bitwise crossover, the answer is generally short, low-order schemas. (See Section 14.4.) But in the case of sequences, it is not immediately obvious what this means. Starkweather et al. [1991] identified three potential kinds of information that might be important for solving an ordering problem and therefore important to preserve through recombination: absolute position in the order, relative ordering (e.g., precedence relations might be important for a scheduling application), and adjacency information (as in the traveling salesman problem). They designed the edge-recombination operator to emphasize adjacency information. The operator is rather complicated, and there are many variants of the originally published operator. A simplified description follows (for details, see Starkweather et al. [1991]). For each pair of individuals to be crossed: (1) construct a table of adjacencies in the parents (see Figure 14.4) and (2) construct one new permutation (offspring) by combining information from the two parents:

- Select one parent at random and assign the first element in its permutation to be the first one in the child.
- Select the second element for the child, as follows: If there is an adjacency common to both parents, then choose that element to be the next one in the child's permutation; if there is an unused adjacency available from one parent, choose it; or if (1) and (2) fail, make a random selection.
- Select the remaining elements in order by repeating step 2.

An example of the edge-recombination operator is shown in Figure 14.4. Although this method has proved effective, it should be noted that it is more expensive to build the adjacency list for each parent and to perform edge recombination operation than it is to use a more standard crossover operator.

A final consideration in the choice of special-purpose operators is the amount of random information that is introduced when the operator is applied. This can be difficult to assess, but it can have a large effect (positive or negative) on the performance of the operator.

14.3.3 Automatic Programming

Genetic algorithms have been used to evolve a special kind of computer program [Koza 1992]. These programs are written in a subset of the programming language Lisp and more recently other languages.

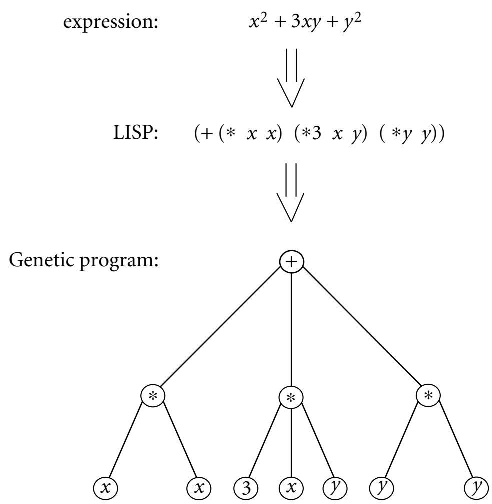


FIGURE 14.5 Tree representation of computer programs: The displayed tree corresponds to the expression $x^2 + 3xy + y^2$. Operators for each expression are displayed as a root, and the operands for each expression are displayed as children. (From Forrest, S. 1993a. *Science* 261:872–878. With permission.)

Lisp programs can naturally be represented as trees (Figure 14.5). Populations of random program trees are generated and evaluated as in the standard genetic algorithm. All other details are similar to those described for binary genetic algorithms with the exception of crossover. Instead of exchanging substrings, **genetic programs** exchange subtrees between individual program trees. This modified form of crossover appears to have many of the same advantages as traditional crossover (such as preserving partial solutions).

Genetic programming has the potential to be extremely powerful, because Lisp is a general-purpose programming language and genetic programming eliminates the need to devise an explicit chromosomal representation. In practice, however, genetic programs are built from subsets of Lisp tailored to particular problem domains, and at this point considerable skill is required to select just the right set of primitives for a particular problem. Although the method has been tested on a wide variety of problems, it has not yet been used extensively in real applications.

The genetic programming method is intriguing because its solutions are so different from human-designed programs for the same problem. Humans try to design elegant and general computer programs, whereas genetic programs are often needlessly complicated, not revealing the underlying algorithm. For example, a human-designed program for computing $\cos 2x$ might be $1 - 2\sin^2 x$, expressed in Lisp as $(-1(*2(*(\sin x)(\sin x))))$, whereas genetic programming discovered the following program (Koza 1992, p. 241):

$$(\sin(-(-2(*x2))(\sin(\sin(\sin(\sin(\sin(\sin(*(\sin(\sin 1))(\sin(\sin 1)))))))))))$$

For anyone who has studied computer programming this is apparently a major drawback because the evolved programs are inelegant, redundant, inefficient, difficult for a human to read, and do not reveal the underlying structure of the algorithm. However, genetic programs do resemble the kinds of ad hoc solutions that evolve in nature through gene duplication, mutation, and modifying structures from one purpose to another. There is some evidence that the junk components of a genetic program sometimes turn out to be useful components in other contexts. Thus, if the genetic programming endeavor is successful, it could revolutionize software design.

14.3.4 Genetic Algorithms for Making Models

The past three examples concentrated on understanding how genetic algorithms can be applied to solve problems. This subsection discusses how the genetic algorithm can be used to model other systems. Genetic algorithms have been employed as models of a wide variety of dynamical processes, including induction in psychology, natural evolution in ecosystems, evolution in immune systems, and imitation in social systems. Making computer models of evolution is somewhat different from many conventional models because the models are highly abstract. The data produced by these models are unlikely to make exact numerical predictions. Rather, they can reveal the conditions under which certain qualitative behaviors are likely to arise — diversity of phenotypes in resource-rich (or poor) environments, cooperation in competitive nonzero-sum games, and so forth. Thus, the models described here are being used to discover qualitative patterns of behavior and, in some cases, critical parameters in which small changes have drastic effects on the outcomes. Such modeling is common in nonlinear dynamics and in artificial intelligence, but it is much less accepted in other disciplines. Here we describe one of these examples: ecological modeling. This exploratory research project is still in an early stage of development. For examples of more mature modeling projects, see Holland et al. [1986] and Axelrod [1986].

The Echo system [Holland 1995] shows how genetic algorithms can be used to model ecosystems. The major differences between Echo and standard genetic algorithms are: (1) there is no explicit fitness function, (2) individuals have local storage (i.e., they consist of more than their genome), (3) the genetic representation is based on a larger alphabet than binary strings, and (4) individuals always have a spatial location. In Echo, fitness evaluation takes place implicitly. That is, individuals in the population (called *agents*) are allowed to make copies of themselves anytime they acquire enough *resources* to replicate their genome. Different resources are modeled by different letters of the alphabet (say, A, B, C, D), and genomes are constructed out of those same letters. These resources can exist independently of the agent's genome, either free in the environment or stored internally by the agent. Agents acquire resources by interacting with other agents through trading relationships and combat. Echo thus relaxes the constraint that an explicit fitness function must return a numerical evaluation of each agent. This **endogenous fitness function** is much closer to the way fitness is assessed in natural settings. In addition to trade and combat, a third form of interaction between agents is mating. Mating provides opportunities for agents to exchange genetic material through crossover, thus creating hybrids. Mating, together with mutation, provides the mechanism for new types of agents to evolve.

Populations in Echo exist on a two-dimensional grid of sites, although other connection topologies are possible. Many agents can cohabit one site, and agents can migrate between sites. Each site is the source of certain renewable resources. On each time step of the simulation, a fixed amount of resources at a site becomes available to the agents located at that site. Different sites can produce different amounts of different resources. For example, one site might produce 10 As and 5 Bs each time step, and its neighbor might produce 5 As, 0 Bs, and 5 Cs. The idea is that an agent will do well (reproduce often) if it is located at a site whose renewable resources match well with its genomic makeup or if it can acquire the relevant resources from other agents at its site.

In preliminary simulations, the Echo system has demonstrated surprisingly complex behaviors, including something resembling a biological arms race (in which two competing species develop progressively more complex offensive and defensive strategies), functional dependencies among different species, trophic cascades, and sensitivity (in terms of the number of different phenotypes) to differing levels of renewable resources. Although the Echo system is still largely untested, it illustrates how the fundamental ideas of genetic algorithms can be incorporated into a system that captures important features of natural ecological systems.

14.4 Mathematical Analysis of Genetic Algorithms

Although there are many problems for which the genetic algorithm can evolve a good solution in reasonable time, there are also problems for which it is inappropriate (such as problems in which it is important to find the exact global optimum). It would be useful to have a mathematical characterization of how

the genetic algorithm works that is predictive. Research on this aspect of genetic algorithms has not produced definitive answers. The domains for which one is likely to choose an adaptive method such as the genetic algorithm are precisely those about which we typically have little analytical knowledge — they are complex, noisy, or dynamic (changing over time). These characteristics make it virtually impossible to predict with certainty how well a particular algorithm will perform on a particular problem instance, especially if the algorithm is stochastic, as is the case with the genetic algorithm. In spite of this difficulty, there are fairly extensive theories about how and why genetic algorithms work in idealized settings.

Analysis of genetic algorithms begins with the concept of a search space. The genetic algorithm can be viewed as a procedure for searching the space of all possible binary strings of fixed length l. Under this interpretation, the algorithm is searching for points in the l-dimensional space $\{0, 1\}^l$ that have high fitness. The search space is identical for all problems of the same size (same l), but the locations of good points will generally differ. The surface defined by the fitness of each point, together with the neighborhood relation imposed by the operators, is sometimes referred to as the **fitness landscape**. The longer the bit strings, corresponding to higher values of l, the larger the search space is, growing exponentially with the length of l. For problems with a sufficiently large l, only a small fraction of this size search space can be examined, and thus it is unreasonable to expect an algorithm to locate the global optimum in the space. A more reasonable goal is to search for good regions of the search space corresponding to regularities in the problem domain. Holland [1975] introduced the notion of a *schema* to explain how genetic algorithms search for regions of high fitness. Schemas are theoretical constructs used to explain the behavior of genetic algorithms, and are not processed directly by the algorithm. The following description of schema processing is excerpted from Forrest and Mitchell [1993b].

A schema is a template, defined over the alphabet $\{0, 1, *\}$, which describes a pattern of bit strings in the search space $\{0, 1\}^l$ (the set of bit strings of length l). For each of the l bit positions, the template either specifies the value at that position (1 or 0), or indicates by the symbol $*$ (referred to as don't care) that either value is allowed.

For example, the two strings A and B have several bits in common. We can use schemas to describe the patterns these two strings share:

A = 100111
B = 010011
$**0{*}11$
$****11$
$**0***$
$**0**1$

A bit string x that matches a schema s's pattern is said to be an *instance* of s; for example, A and B are both instances of the schemas just shown. In schemas, 1s and 0s are referred to as *defined bits*; the *order* of a schema is the number of defined bits in that schema, and the *defining length* of a schema is the distance between the leftmost and rightmost defined bits in the string. For example, the defining length of $**0**1$ is 3.

Schemas define hyperplanes in the search space $\{0, 1\}^l$. Figure 14.6 shows four hyperplanes, corresponding to the schemas $0****$, $1****$, $*0***$, and $*1***$. Any point in the space is simultaneously an instance

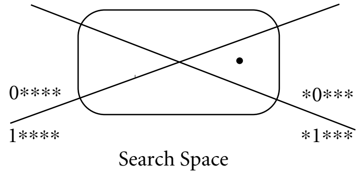


FIGURE 14.6 Schemas define hyperplanes in the search space. (From Forrest, S. and Mitchell, M. 1993b. *Machine Learning* 13:285–319. With permission.)

of two of these schemas. For example, the point shown in Figure 14.6 is an instance of both 1∗∗∗∗ and ∗0∗∗∗ (and also of 10∗∗∗).

The fitness of any bit string in the population gives some information about the average fitness of the 2^l different schemas of which it is an instance, and so an explicit evaluation of a population of M individual strings is also an implicit evaluation of a much larger number of schemas. This is referred to as implicit parallelism. At the explicit level the genetic algorithm searches through populations of bit strings, but the genetic algorithm's search can also be interpreted as an implicit schema sampling process. Feedback from the fitness function, combined with selection and recombination, biases the sampling procedure over time away from those schemas that give negative feedback (low average fitness) and toward those that give positive feedback (high average fitness). Ultimately, the search procedure should identify regularities, or patterns, in the environment that lead to high fitness. Because the space of possible patterns is larger than the space of possible individuals (3^l vs. 2^l), implicit parallelism is potentially advantageous.

An important theoretical result about genetic algorithms is the Schema Theorem [Holland 1975, Goldberg 1989], which states that the observed best schemas will on average be allocated an exponentially increasing number of samples in the next generation. Figure 14.7 illustrates the rapid convergence on fit schemas by the genetic algorithm. This strong convergence property of the genetic algorithm is a two-edged

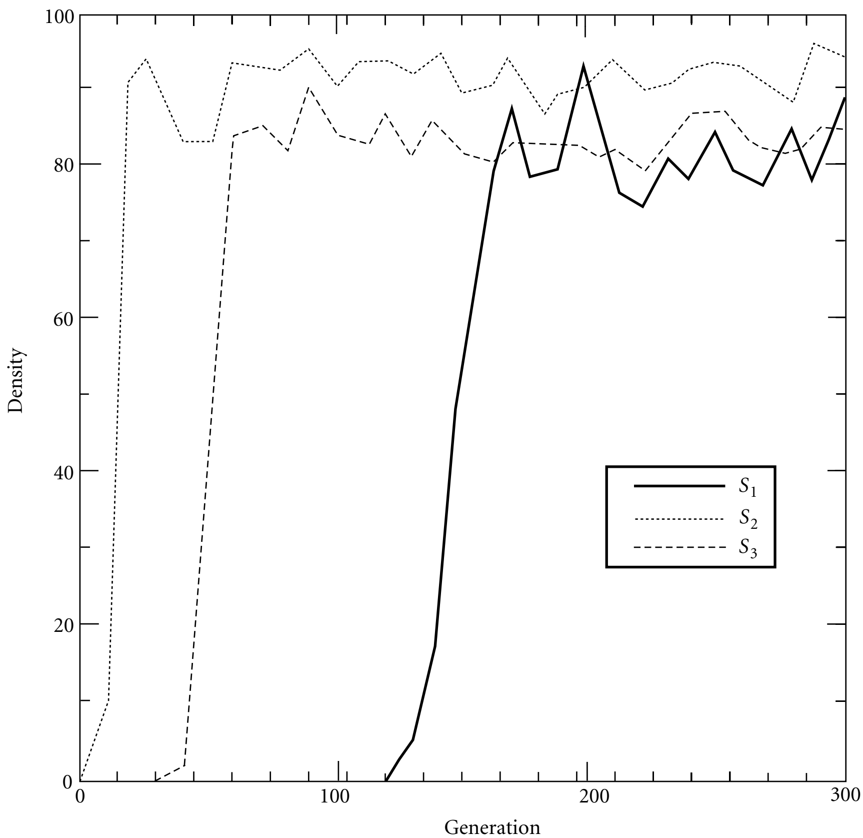


FIGURE 14.7 Schema frequencies over time. The graph plots schema frequencies in the population over time for three schemas:

$$s_1 = 1111111111111111{*};$$
$$s_2 = {*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}1111111111111111{*};$$
$$s_3 = {*}111111111111111111111111111111.$$

The function plotted was a royal road function [Forrest and Mitchell 1993a] in which the optimum value is the string of all 1s. (From Forrest, S. 1993a. *Science* 261:872–878. With permission.)

sword. On the one hand, the fact that the genetic algorithm can close in on a fit part of the space very quickly is a powerful property; on the other hand, because the genetic algorithm always operates on finite-size populations, there is inherently some sampling error in the search, and in some cases the genetic algorithm can magnify a small sampling error, causing premature convergence on local optima.

According to the **building blocks hypothesis** [Holland 1975, Goldberg 1989], the genetic algorithm initially detects biases toward higher fitness in some low-order schemas (those with a small number of defined bits), and converges on this part of the search space. Over time, it detects biases in higher-order schemas by combining information from low-order schemas via crossover, and eventually it converges on a small region of the search space that has high fitness. The building blocks hypothesis states that this process is the source of the genetic algorithm's power as a search and optimization method. If this hypothesis about how genetic algorithms work is correct, then crossover is of primary importance, and it distinguishes genetic algorithms from other similar methods, such as simulated annealing and greedy algorithms. A number of authors have questioned the adequacy of the building blocks hypothesis as an explanation for how genetic algorithms work and there are several active research efforts studying schema processing in genetic algorithms. Nevertheless, the explanation of schemas and recombination that I have just described stands as the most common account of why genetic algorithms perform as they do.

There are several other approaches to analyzing mathematically the behavior of genetic algorithms: models developed for population genetics, algebraic models, signal-to-noise analysis, landscape analysis, statistical mechanics, Markov chains, and methods based on probably approximately correct (PAC) learning. This work extends and refines the schema analysis just given and in some cases challenges the claim that recombination through crossover is an important component of genetic algorithm performance. See Further Information section for additional reading.

14.5 Research Issues and Summary

The idea of using evolution to solve difficult problems and to model natural phenomena is promising. The genetic algorithms that I have described in this chapter are one of the first steps in this direction. Necessarily, they have abstracted out much of the richness of biology, and in the future we can expect a wide variety of evolutionary systems based on the principles of genetic algorithms but less closely tied to these specific mechanisms. For example, more elaborate representation techniques, including those that use complex genotype-to-phenotype mappings and increasing use of nonbinary alphabets can be expected. Endogenous fitness functions, similar to the one described for Echo, may become more common, as well as dynamic and coevolutionary fitness functions. More generally, biological mechanisms of all kinds will be incorporated into computational systems, including nervous systems, embryology, parasites, viruses, and immune systems.

From an algorithmic perspective, genetic algorithms join a broader class of stochastic methods for solving problems. An important area of future research is to understand carefully how these algorithms relate to one another and which algorithms are best for which problems. This is a difficult area in which to make progress. Controlled studies on idealized problems may have little relevance for practical problems, and benchmarks on specific problem instances may not apply to other instances. In spite of these impediments, this is an important direction for future research.

Acknowledgments

The author gratefully acknowledges support from the National Science Foundation (Grant IRI-9157644), the Office of Naval Research (Grant N00014-95-1-0364), ATR Human Information Processing Research Laboratories, and the Santa Fe Institute. Ron Hightower prepared Figure 14.2.

Significant portions of this chapter are excerpted with permission from Forrest, S. 1993. Genetic algorithms: principles of adaption applied to computation. *Science* 261 (Aug. 13):872–878. © 1993 American Association for the Advancement of Science.

Defining Terms

Building blocks hypothesis: The hypothesis that the genetic algorithm searches by first detecting biases toward higher fitness in some low-order schemas (those with a small number of defined bits) and converging on this part of the search space. Over time, it then detects biases in higher-order schemas by combining information from low-order schemas via crossover and eventually converges on a small region of the search space that has high fitness. The building blocks hypothesis states that this process is the source of the genetic algorithm's power as a search and optimization method [Holland 1975, Goldberg 1989].

Chromosome: A string of symbols (usually in bits) that contains the genetic information about an individual. The chromosome is interpreted by the fitness function to produce an evaluation of the individual's fitness.

Crossover: An operator for producing new individuals from two parent individuals. The operator works by exchanging substrings between the two individuals to obtain new offspring. In some cases, both offspring are passed to the new generation; in others, one is arbitrarily chosen to be passed on; the number of crossover points can be restricted to one per pair, two per pair, or N per pair.

Edge recombination: A special-purpose crossover operator designed to be used with permutation representations for sequencing problems. The edge-recombination operator attempts to preserve adjacencies between neighboring elements in the parent permutations [Starkweather et al. 1991].

Endogenous fitness function: Fitness is not assessed explicitly using a fitness function. Some other criterion for reproduction is adopted. For example, individuals might be required to accumulate enough internal resources to copy themselves before they can reproduce. Individuals who can gather resources efficiently would then reproduce frequently and their traits would become more prevalent in the population.

Fitness function: Each individual is tested empirically in an environment, receiving a numerical evaluation of its merit, assigned by a fitness function F. The environment can be almost anything — another computer simulation, interactions with other individuals in the population, actions in the physical world (by a robot for example), or a human's subjective judgment.

Fitness landscape: The surface defined by the fitness of each point in the search space, together with the neighborhood relation imposed by the operators.

Generation: One iteration, or time step, of the genetic algorithm. New generations can be produced either synchronously, so that the old generation is completely replaced (the time step model), or asynchronously, so that generations overlap. In the asynchronous case, generations are defined in terms of some fixed number of fitness-function evaluations.

Genetic programs: A form of genetic algorithm that uses a tree-based representation. The tree represents a program that can be evaluated, for example, an S-expression.

Genotype: The string of symbols, usually bits, used to represent an individual. Each bit position (set to 1 or 0) represents one gene. The term bit string in this context refers both to genotypes and to the individuals that they define.

Individuals: The structures that are evolved by the genetic algorithm. They can represent solutions to problems, strategies for playing games, visual images, or computer programs. Typically, each individual consists only of its genetic material, which is organized into one (haploid) chromosome.

Mutation: An operator for varying an individual. In mutation, individual bits are flipped probabilistically in individuals selected for reproduction. In representations other than bit strings, mutation is redefined to an appropriate smallest unit of change. For example, in permutation representations, mutation is often defined to be the swap of two neighboring elements in the permutation; in real-valued representations, mutation can be a creep operator that perturbs the real number up or down some small increment.

Schema: A theoretical construct used to explain the behavior of genetic algorithms. Schemas are not processed directly by the algorithm. Schemas are coordinate hyperplanes in the search space of strings.

Selection: Some individuals are more fit than others (better suited to their environment). These individuals have more offspring, that is, they are selected for reproduction. Selection is implemented by eliminating low-fitness individuals from the population, and inheritance is implemented by making multiple copies of high-fitness individuals.

References

Axelrod, R. 1986. An evolutionary approach to norms. *Am. Political Sci. Rev.* 80 (Dec).

Back, T. and Schwefel, H. P. 1993. An overview of evolutionary algorithms. *Evolutionary Comput.* 1:1–23.

Belew, R. K. and Booker, L. B., eds. 1991. *Proc. 4th Int. Conf. Genet. Algorithms*. July. Morgan Kaufmann, San Mateo, CA.

Booker, L. B., Riolo, R. L., and Holland, J. H. 1989. Learning and representation in classifier systems. *Art. Intelligence* 40:235–282.

Box, G. E. P. 1957. Evolutionary operation: a method for increasing industrial productivity. *J. R. Stat. Soc.* 6(2):81–101.

Davis, L., ed. 1991. *The Genetic Algorithms Handbook*. Van Nostrand Reinhold, New York.

DeJong, K. A. 1975. *An analysis of the behavior of a class of genetic adaptive systems*. Ph.D. thesis, University of Michigan, Ann Arbor.

DeJong, K. A. 1990a. Genetic-algorithm-based learning. *Machine Learning* 3:611–638.

DeJong, K. A. 1990b. Introduction to second special issue on genetic algorithms. *Machine Learning*. 5(4):351–353.

Eshelman, L. J., ed. 1995. *Proc. 6th Int. Conf. Genet. Algorithms*. Morgan Kaufmann, San Francisco.

Filho, J. L. R., Treleaven, P. C., and Alippi, C. 1994. Genetic-algorithm programming environments. *Computer* 27(6):28–45.

Fogel, L. J., Owens, A. J., and Walsh, M. J. 1966. *Artificial Intelligence Through Simulated Evolution*. Wiley, New York.

Fonseca, C. M. and Fleming, P. J. 1995. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Comput.* 3(1):1–16.

Forrest, S. 1993a. Genetic algorithms: principles of adaptation applied to computation. *Science* 261:872–878.

Forrest, S., ed. 1993b. *Proc. Fifth Int. Conf. Genet. Algorithms*. Morgan Kaufmann, San Mateo, CA.

Forrest, S. and Mitchell, M. 1993a. Towards a stronger building-blocks hypothesis: effects of relative building-block fitness on ga performance. In *Foundations of Genetic Algorithms*, Vol. 2, L. D. Whitley, ed., pp. 109–126. Morgan Kaufmann, San Mateo, CA.

Forrest, S. and Mitchell, M. 1993b. What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. *Machine Learning* 13(2/3).

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, Reading, MA.

Grefenstette, J. J. 1985. *Proc. Int. Conf. Genet. Algorithms Appl.* NCARAI and Texas Instruments.

Grefenstette, J. J. 1987. *Proc. 2nd Int. Conf. Genet. Algorithms*. Lawrence Erlbaum, Hillsdale, NJ.

Hillis, W. D. 1990. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D* 42:228–234.

Holland, J. H. 1962. Outline for a logical theory of adaptive systems. *J. ACM* 3:297–314.

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI; 1992. 2nd ed. MIT Press, Cambridge, MA.

Holland, J. H. 1992. Genetic algorithms. *Sci. Am.*, pp. 114–116.

Holland, J. H. 1995. *Hidden Order: How Adaptation Builds Complexity*. Addison–Wesley, Reading, MA.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., and Thagard, P. 1986. *Induction: Processes of Inference, Learning, and Discovery*. MIT Press, Cambridge, MA.

Koza, J. R. 1992. *Genetic Programming*. MIT Press, Cambridge, MA.

Männer, R. and Manderick, B., eds. 1992. *Parallel Problem Solving From Nature 2*. North Holland, Amsterdam.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

Mitchell, M. and Forrest, S. 1994. Genetic algorithms and artificial life. *Artif. Life* 1(3):267–289; reprinted 1995. In *Artificial Life: An Overview*, C. G. Langton, ed. MIT Press, Cambridge, MA.

Mühlenbein, H., Gorges-Schleuter, M., and Kramer, O. 1988. *Parallel Comput.* 6:65–88.

Rawlins, G., ed. 1991. *Foundations of Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA.

Schaffer, J. D., ed. 1989. *Proc. 3rd Int. Conf. Genet. Algorithms*. Morgan Kaufmann, San Mateo, CA.

Schaffer, J. D., Whitley, D., and Eshelman, L. J. 1992. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *Int. Workshop Combinations Genet. Algorithms Neural Networks*, L. D. Whitley and J. D. Schaffer, eds., pp. 1–37. IEEE Computer Society Press, Los Alamitos, CA.

Schwefel, H. P. and Männer, R., eds. 1990. Parallel problem solving from nature. *Lecture Notes in Computer Science*. Springer–Verlag, Berlin.

Srinivas, M. and Patnaik, L. M. 1994. Genetic algorithms: a survey. *Computer* 27(6):17–27.

Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., and Whitley, C. 1991. A comparison of genetic sequencing operators. In *4th Int. Conf. Genet. Algorithms*, R. K. Belew and L. B. Booker, eds., pp. 69–76. Morgan Kaufmann, Los Altos, CA.

Thomas, E. V. 1993. Frequency Selection Using Genetic Algorithms. *Sandia National Lab. Tech. Rep.* SAND93-0010, Albuquerque, NM.

Whitley, L. D., ed. 1993. *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Mateo, CA.

Whitley, L. D. and Vose, M., eds. 1995. *Foundations of Genetic Algorithms 3*. Morgan Kaufmann, San Francisco.

Further Information

Review articles on genetic algorithms include Booker et al. [1989], Holland [1992], Forrest [1993a], Mitchell and Forrest [1994], Srinivas and Patnaik [1994] and Filho et al. [1994]. Books that describe the theory and practice of genetic algorithms in greater detail include Holland [1975], Goldberg [1989], Davis [1991], Koza [1992], Holland et al. [1986], and Mitchell [1996]. Holland [1975] was the first book-length description of genetic algorithms, and it contains much of the original insight about the power and breadth of adaptive algorithms. The 1992 reprinting contains interesting updates by Holland. However, Goldberg [1989], Davis [1991], and Mitchell [1996] are more accessible introductions to the basic concepts and implementation issues. Koza [1992] describes genetic programming and Holland et al. [1986] discuss the relevance of genetic algorithms to cognitive modeling.

Current research on genetic algorithms is reported many places, including the Proceedings of the International Conference on Genetic Algorithms [Grefenstette 1985, 1987, Schaffer 1989, Belew and Booker 1991, Forrest 1993b, Eshelman 1995], the proceedings of conferences on Parallel Problem Solving from Nature [Schwefel and Männer 1990, Männer and Manderick 1992], and the workshops on Foundations of Genetic Algorithms [Rawlins 1991, Whitley 1993, Whitley and Vose 1995]. Finally, the artificial-life literature contains many interesting papers about genetic algorithms.

There are several archival journals that publish articles about genetic algorithms. These include *Evolutionary Computation* (a journal devoted to GAs), *Complex Systems*, *Machine Learning*, *Adaptive Behavior*, and *Artificial Life*.

Information about genetic algorithms activities, public domain packages, etc., is maintained through the WWW at URL http://www.aic.nrl.navy.mil/galist/ or through anonymous ftp at ftp.aic.nrl.navy.mil [192.26.18.68] in/pub/galist.

15
Combinatorial Optimization

Vijay Chandru
Indian Institute of Science

M. R. Rao
Indian Institute of Management

15.1 Introduction

Bin packing, routing, scheduling, layout, and network design are generic examples of combinatorial optimization problems that often arise in computer engineering and decision support. Unfortunately, almost all interesting generic classes of combinatorial optimization problems are $\mathcal{NP}$-hard. The scale at which these problems arise in applications and the explosive exponential complexity of the search spaces preclude the use of simplistic enumeration and search techniques. Despite the worst-case intractability of combinatorial optimization, in practice we are able to solve many large problems and often with off-the-shelf software. Effective software for combinatorial optimization is usually problem specific and based on sophisticated algorithms that combine approximation methods with search schemes and that exploit mathematical (and not just syntactic) structure in the problem at hand.

Multidisciplinary interests in combinatorial optimization have led to several fairly distinct paradigms in the development of this subject. Each paradigm may be thought of as a particular combination of a *representation scheme* and a *methodology* (see Table 15.1). The most established of these, the **integer programming** paradigm, uses implicit algebraic forms (linear constraints) to represent combinatorial

TABLE 15.1 Paradigms in Combinatorial Optimization

Paradigm	Representation	Methodology
Integer programming	Linear constraints, Linear objective, Integer variables	Linear programming and extensions
Search	State space, Discrete control	Dynamic programming, $\mathcal{A}^*$
Local improvement	Neighborhoods Fitness functions	Hill climbing, Simulated annealing, Tabu search, Genetic algorithms
Constraint logic programming	Horn rules	Resolution, constraint solvers

optimization and **linear programming** and its extensions as the workhorses in the design of the solution algorithms. It is this paradigm that forms the central theme of this chapter.

Other well known paradigms in combinatorial optimization are **search, local improvement,** and **constraint logic programming.** Search uses state-space representations and partial enumeration techniques such as $\mathcal{A}^*$ and dynamic programming. Local improvement requires only a representation of neighborhood in the solution space, and methodologies vary from simple hill climbing to the more sophisticated techniques of simulated annealing, tabu search, and genetic algorithms. Constraint logic programming uses the syntax of Horn rules to represent combinatorial optimization problems and uses resolution to orchestrate the solution of these problems with the use of domain-specific constraint solvers. Whereas integer programming was developed and nurtured by the mathematical programming community, these other paradigms have been popularized by the artificial intelligence community.

An abstract formulation of combinatorial optimization is

$$\text{(CO)} \quad \min\{f(I) : I \in \mathcal{I}\}$$

where $\mathcal{I}$ is a collection of subsets of a finite ground set $E = \{e_1, e_2, \dots, e_n\}$ and f is a criterion (objective) function that maps 2^E (the power set of E) to the reals. A **mixed integer linear program** (MILP) is of the form

$$\text{(MILP)} \quad \min_{x \in \Re^n}\{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b},\ \mathbf{x}_j \text{ integer } \forall\, j \in J\}$$

which seeks to minimize a linear function of the decision vector **x** subject to linear inequality constraints and the requirement that a subset of the decision variables is integer valued. This model captures many variants. If $J = \{1, 2, \dots, n\}$, we say that the integer program is *pure*, and *mixed* otherwise. Linear equations and bounds on the variables can be easily accommodated in the inequality constraints. Notice that by adding in inequalities of the form $0 \leq \mathbf{x}_j \leq 1$ for a $j \in J$ we have forced $\mathbf{x}_j$ to take value 0 or 1. It is such Boolean variables that help capture combinatorial optimization problems as special cases of MILP.

Pure integer programming with variables that take arbitrary integer values is a class which has strong connections to number theory and particularly the geometry of numbers and Presburgher arithmetic. Although this is a fascinating subject with important applications in cryptography, in the interests of brevity we shall largely restrict our attention to MILP where the integer variables are Boolean.

The fact that mixed integer linear programs subsume combinatorial optimization problems follows from two simple observations. The first is that a collection $\mathcal{I}$ of subsets of a finite ground set E can always be represented by a corresponding collection of incidence vectors, which are $\{0, 1\}$-vectors in $\Re^E$. Further, arbitrary nonlinear functions can be represented via piecewise linear approximations by using linear constraints and mixed variables (continuous and Boolean).

The next section contains a primer on linear inequalities, polyhedra, and linear programming. These are the tools we will need to analyze and solve integer programs. Section 15.4, is a testimony to the earlier

cryptic comments on how integer programs model combinatorial optimization problems. In addition to working a number of examples of such integer programming formulations, we shall also review a formal representation theory of (Boolean) mixed integer linear programs.

With any mixed integer program we associate a **linear programming relaxation** obtained by simply ignoring the integrality restrictions on the variables. The point being, of course, that we have polynomial-time (and practical) algorithms for solving linear programs. Thus, the linear programming relaxation of (MILP) is given by

$$\text{(LP)} \qquad \min_{x \in \Re^n} \{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b}\}$$

The thesis underlying the integer linear programming approach to combinatorial optimization is that this linear programming relaxation retains enough of the structure of the combinatorial optimization problem to be a useful weak representation. In Section 15.5 we shall take a closer look at this thesis in that we shall encounter special structures for which this relaxation is *tight*. For general integer programs, there are several alternative schemes for generating linear programming relaxations with varying qualities of approximation. A general principle is that we often need to disaggregate integer formulations to obtain higher quality linear programming relaxations. To solve such huge linear programs we need specialized techniques of large-scale linear programming. These aspects will be the content of Section 15.3.

The reader should note that the focus in this chapter is on solving hard combinatorial optimization problems. We catalog the special structures in integer programs that lead to tight linear programming relaxations (Section 15.5) and hence to polynomial-time algorithms. These include structures such as network flows, matching, and matroid optimization problems. Many hard problems actually have pieces of these nice structures embedded in them. Practitioners of combinatorial optimization have always used insights from special structures to devise strategies for hard problems.

The computational art of integer programming rests on useful interplays between search methodologies and linear programming relaxations. The paradigms of branch and bound and branch and cut are the two enormously effective partial enumeration schemes that have evolved at this interface. These will be discussed in Section 15.6. It may be noted that all general purpose integer programming software available today uses one or both of these paradigms.

The inherent complexity of integer linear programming has led to a long-standing research program in approximation methods for these problems. Linear programming relaxation and Lagrangian relaxation are two general approximation schemes that have been the real workhorses of computational practice. Primal–dual strategies and semidefinite relaxations are two recent entrants that appear to be very promising. Section 15.7 of this chapter reviews these developments in the approximation of combinatorial optimization problems.

We conclude the chapter with brief comments on future prospects in combinatorial optimization from the algebraic modeling perspective.

15.2 A Primer on Linear Programming

Polyhedral combinatorics is the study of embeddings of combinatorial structures in Euclidean space and their algebraic representations. We will make extensive use of some standard terminology from polyhedral theory. Definitions of terms not given in the brief review below can be found in Nemhauser and Wolsey [1988].

A (convex) **polyhedron** in $\Re^n$ can be algebraically defined in two ways. The first and more straightforward definition is the *implicit* representation of a polyhedron in $\Re^n$ as the solution set to a finite system of linear inequalities in n variables. A single linear inequality $\mathbf{ax} \leq a_0$; $\mathbf{a} \neq \mathbf{0}$ defines a *half-space* of $\Re^n$. Therefore, geometrically a polyhedron is the intersection set of a finite number of half-spaces.

A *polytope* is a bounded polyhedron. Every polytope is the convex closure of a finite set of points. Given a set of points whose convex combinations generate a polytope, we have an explicit or *parametric* algebraic representation of it. A *polyhedral cone* is the solution set of a system of homogeneous linear inequalities.

Every (polyhedral) cone is the conical or positive closure of a finite set of vectors. These generators of the cone provide a parametric representation of the cone. And finally, a polyhedron can be alternatively defined as the Minkowski sum of a polytope and a cone. Moving from one representation of any of these polyhedral objects to another defines the essence of the computational burden of polyhedral combinatorics. This is particularly true if we are interested in *minimal* representations.

A set of points $\mathbf{x}^1, \ldots, \mathbf{x}^m$ is *affinely independent* if the unique solution of $\sum_{i=1}^m \lambda_i \mathbf{x}^i = 0$, $\sum_{i=1}^m \lambda_i = 0$ is $\lambda_i = 0$ for $i = 1, \ldots, m$. Note that the maximum number of affinely independent points in $\Re^n$ is $n + 1$. A polyhedron P is of *dimension* k, dim $P = k$, if the maximum number of affinely independent points in P is $k + 1$. A polyhedron $P \subseteq \Re^n$ of dimension n is called *full dimensional.* An inequality $\mathbf{ax} \leq a_0$ is called *valid* for a polyhedron P if it is satisfied by all $\mathbf{x}$ in P. It is called *supporting* if in addition there is an $\tilde{\mathbf{x}}$ in P that satisfies $\mathbf{a}\tilde{\mathbf{x}} = a_0$. A *face* of the polyhedron is the set of all $\mathbf{x}$ in P that also satisfies a valid inequality as an equality. In general, many valid inequalities might represent the same face. Faces other than P itself are called *proper.* A *facet* of P is a maximal nonempty and proper face. A facet is then a face of P with a dimension of dim $P - 1$. A face of dimension zero, i.e., a point v in P that is a face by itself, is called an **extreme point** of P. The extreme points are the elements of P that cannot be expressed as a strict convex combination of two distinct points in P. For a full-dimensional polyhedron, the valid inequality representing a facet is unique up to multiplication by a positive scalar, and facet-inducing inequalities give a minimal implicit representation of the polyhedron. Extreme points, on the other hand, give rise to minimal parametric representations of polytopes.

The two fundamental problems of linear programming (which are polynomially equivalent) follow:

- *Solvability.* This is the problem of checking if a system of linear constraints on real (rational) variables is solvable or not. Geometrically, we have to check if a polyhedron, defined by such constraints, is nonempty.
- *Optimization.* This is the problem (LP) of optimizing a linear objective function over a polyhedron described by a system of linear constraints.

Building on polarity in cones and polyhedra, duality in linear programming is a fundamental concept which is related to both the complexity of linear programming and to the design of algorithms for solvability and optimization. We will encounter the solvability version of duality (called Farkas Lemma) while discussing the Fourier elimination technique subsequently. Here we will state the main duality results for optimization. If we take the *primal* linear program to be

$$(P) \qquad \min_{x \in \Re^n} \{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b}\}$$

there is an associated *dual* linear program

$$(D) \qquad \max_{y \in \Re^m} \{\mathbf{b}^T\mathbf{y} : A^T\mathbf{y} = \mathbf{c}^T,\ \mathbf{y} \geq \mathbf{0}\}$$

and the two problems satisfy the following:

1. For any $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ feasible in (P) and (D) (i.e., they satisfy the respective constraints), we have $\mathbf{c}\hat{\mathbf{x}} \geq \mathbf{b}^T\hat{\mathbf{y}}$ (weak duality). Consequently, (P) has a finite optimal solution if and only if (D) does.
2. The pair $\mathbf{x}^*$ and $\mathbf{y}^*$ are optimal solutions for (P) and (D), respectively, if and only if $\mathbf{x}^*$ and $\mathbf{y}^*$ are feasible in (P) and (D) (i.e., they satisfy the respective constraints) and $\mathbf{cx}^* = \mathbf{b}^T\mathbf{y}^*$ (strong duality).
3. The pair $\mathbf{x}^*$ and $\mathbf{y}^*$ are optimal solutions for (P) and (D), respectively, if and only if $\mathbf{x}^*$ and $\mathbf{y}^*$ are feasible in (P) and (D) (i.e., they satisfy the respective constraints) and $(A\mathbf{x}^* - \mathbf{b})^T\mathbf{y}^* = 0$ (complementary slackness).

The strong duality condition gives us a good stopping criterion for optimization algorithms. The complementary slackness condition, on the other hand, gives us a constructive tool for moving from dual

to primal solutions and vice versa. The weak duality condition gives us a technique for obtaining lower bounds for minimization problems and upper bounds for maximization problems.

Note that the properties just given have been stated for linear programs in a particular form. The reader should be able to check that if, for example, the primal is of the form

$$(P') \quad \min_{\mathbf{x}\in\Re^n}\{\mathbf{cx} : A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$$

then the corresponding dual will have the form

$$(D') \quad \max_{\mathbf{y}\in\Re^m}\{\mathbf{b}^T\mathbf{y} : A^T\mathbf{y} \leq \mathbf{c}^T\}$$

The tricks needed for seeing this are that any equation can be written as two inequalities, an unrestricted variable can be substituted by the difference of two nonnegatively constrained variables, and an inequality can be treated as an equality by adding a nonnegatively constrained variable to the lesser side. Using these tricks, the reader could also check that duality in linear programming is involutory (i.e., the dual of the dual is the primal).

15.2.1 Algorithms for Linear Programming

We will now take a quick tour of some algorithms for linear programming. We start with the classical technique of Fourier, which is interesting because of its really simple syntactic specification. It leads to simple proofs of the duality principle of linear programming (solvability) that has been alluded to. We will then review the simplex method of linear programming, a method that has been finely honed over almost five decades. We will spend some time with the ellipsoid method and, in particular, with the polynomial equivalence of solvability (optimization) and separation problems, for this aspect of the ellipsoid method has had a major impact on the identification of many tractable classes of combinatorial optimization problems. We conclude the primer with a description of Karmarkar's [1984] breakthrough, which was an important landmark in the brief history of linear programming. A noteworthy role of interior point methods has been to make practical the theoretical demonstrations of tractability of various aspects of linear programming, including solvability and optimization, that were provided via the ellipsoid method.

15.2.1.1 Fourier's Scheme for Linear Inequalities

Constraint systems of linear *inequalities* of the form $A\mathbf{x} \leq \mathbf{b}$, where A is an $m \times n$ matrix of real numbers, are widely used in mathematical models. Testing the solvability of such a system is equivalent to linear programming.

Suppose we wish to eliminate the first variable $\mathbf{x}_1$ from the system $A\mathbf{x} \leq \mathbf{b}$. Let us denote

$$I^+ = \{i : A_{i1} > 0\} \qquad I^- = \{i : A_{i1} < 0\} \qquad I^0 = \{i : A_{i1} = 0\}$$

Our goal is to create an equivalent system of linear inequalities $\tilde{A}\tilde{\mathbf{x}} \leq \tilde{\mathbf{b}}$ defined on the variables $\tilde{\mathbf{x}} = (\mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n)$:

- If I^+ is empty then we can simply delete all the inequalities with indices in I^- since they can be trivially satisfied by choosing a large enough value for $\mathbf{x}_1$. Similarly, if I^- is empty we can discard all inequalities in I^+.
- For each $k \in I^+$, $l \in I^-$ we add $-A_{l1}$ times the inequality $A_k\mathbf{x} \leq \mathbf{b}_k$ to A_{k1} times $A_l\mathbf{x} \leq \mathbf{b}_l$. In these new inequalities the coefficient of $\mathbf{x}_1$ is wiped out, that is, $\mathbf{x}_1$ is eliminated. Add these new inequalities to those already in I^0.
- The inequalities $\{\tilde{A}_{i1}\tilde{\mathbf{x}} \leq \tilde{\mathbf{b}}_i\}$ for all $i \in I^0$ represent the equivalent system on the variables $\tilde{\mathbf{x}} = (\mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n)$.

Repeat this construction with $\tilde{A}\tilde{\mathbf{x}} \leq \tilde{\mathbf{b}}$ to eliminate $\mathbf{x}_2$ and so on until all variables are eliminated. If the resulting $\tilde{\mathbf{b}}$ (after eliminating $\mathbf{x}_n$) is nonnegative, we declare the original (and intermediate) inequality systems as being consistent. Otherwise,* $\tilde{\mathbf{b}} \not\geq 0$ and we declare the system inconsistent.

As an illustration of the power of elimination as a tool for theorem proving, we show now that Farkas Lemma is a simple consequence of the correctness of Fourier elimination. The lemma gives a direct proof that solvability of linear inequalities is in $\mathcal{NP} \bigcap co\mathcal{NP}$.

FARKAS LEMMA 15.1 (Duality in Linear Programming: Solvability). *Exactly one of the alternatives*

$$I. \quad \exists\, \mathbf{x} \in \Re^n \; : \; A\mathbf{x} \leq \mathbf{b}$$

$$II. \quad \exists\, \mathbf{y} \in \Re^m_+ \; : \; \mathbf{y}^t A = \mathbf{0},\; \mathbf{y}^t\mathbf{b} < 0$$

is true for any given real matrices $A, \mathbf{b}$.

***Proof* 15.1** Let us analyze the case when Fourier elimination provides a proof of the inconsistency of a given linear inequality system $A\mathbf{x} \leq \mathbf{b}$. The method clearly converts the given system into $RA\mathbf{x} \leq R\mathbf{b}$ where RA is zero and $R\mathbf{b}$ has at least one negative component. Therefore, there is some row of R, say, $\mathbf{r}$, such that $\mathbf{r}A = \mathbf{0}$ and $\mathbf{rb} < 0$. Thus $\neg I$ implies II. It is easy to see that I and II cannot both be true for fixed $A, \mathbf{b}$. □

In general, the Fourier elimination method is quite inefficient. Let k be any positive integer and n the number of variables be $2^k + k + 2$. If the input inequalities have left-hand sides of the form $\pm\mathbf{x}_r \pm \mathbf{x}_s \pm \mathbf{x}_t$ for all possible $1 \leq r < s < t \leq n$, it is easy to prove by induction that after k variables are eliminated, by Fourier's method, we would have at least $2^{n/2}$ inequalities. The method is therefore exponential in the worst case, and the explosion in the number of inequalities has been noted, in practice as well, on a wide variety of problems. We will discuss the central idea of minimal generators of the projection cone that results in a much improved elimination method.

First, let us identify the set of variables to be eliminated. Let the input system be of the form

$$P = \{(\mathbf{x}, \mathbf{u}) \in \Re^{n_1+n_2} \;\mid\; A\mathbf{x} + B\mathbf{u} \leq \mathbf{b}\}$$

where $\mathbf{u}$ is the set to be eliminated. The projection of P onto $\mathbf{x}$ or equivalently the effect of eliminating the $\mathbf{u}$ variables is

$$P_{\mathbf{x}} = \{\mathbf{x} \in \Re^{n_1} \;\mid\; \exists \mathbf{u} \in \Re^{n_2} \text{ such that } A\mathbf{x} + B\mathbf{u} \leq \mathbf{b}\}$$

Now W, the *projection cone* of P, is given by

$$W = \{\mathbf{w} \in \Re^m \;\mid\; \mathbf{w}B = \mathbf{0},\; \mathbf{w} \geq \mathbf{0}\}$$

A simple application of Farkas Lemma yields a description of $P_{\mathbf{x}}$ in terms of W.

PROJECTION LEMMA 15.2 *Let G be any set of generators (e.g., the set of extreme rays) of the cone W. Then* $P_{\mathbf{x}} = \{\mathbf{x} \in \Re^{n_1} | (\mathbf{g}A)x \leq \mathbf{g}b\, \forall\, \mathbf{g} \in G\}$.

The lemma, sometimes attributed to Černikov [1961], reduces the computation of $P_{\mathbf{x}}$ to enumerating the extreme rays of the cone W or equivalently the extreme points of the polytope $W \cap \{\mathbf{w} \in \Re^m | \sum_{i=1}^m \mathbf{w}_i = 1\}$.

*Note that the final $\tilde{\mathbf{b}}$ may not be defined if all of the inequalities are deleted by the monotone sign condition of the first step of the construction described. In such a situation, we declare the system $A\mathbf{x} \leq \mathbf{b}$ *strongly consistent* since it is consistent for any choice of $\mathbf{b}$ in $\Re^m$. To avoid making repeated references to this exceptional situation, let us simply assume that it does not occur. The reader is urged to verify that this assumption is indeed benign.

15.2.1.2 Simplex Method

Consider a polyhedron $\mathcal{K} = \{\mathbf{x} \in \Re^n : A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$. Now $\mathcal{K}$ cannot contain an infinite (in both directions) line since it is lying within the nonnegative orthant of $\Re^n$. Such a polyhedron is called a *pointed* polyhedron. Given a pointed polyhedron $\mathcal{K}$ we observe the following:

- If $\mathcal{K} \neq \emptyset$, then $\mathcal{K}$ has at least one extreme point.
- If $\min\{\mathbf{cx} : A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ has an optimal solution, then it has an optimal extreme point solution.

These observations together are sometimes called the fundamental theorem of linear programming since they suggest simple finite tests for both solvability and optimization. To generate all extreme points of $\mathcal{K}$, in order to find an optimal solution, is an impractical idea. However, we may try to run a partial search of the space of extreme points for an optimal solution. A simple local improvement search strategy of moving from extreme point to adjacent extreme point until we get to a local optimum is nothing but the simplex method of linear programming. The local optimum also turns out to be a global optimum because of the convexity of the polyhedron $\mathcal{K}$ and the linearity of the objective function $\mathbf{cx}$.

The simplex method walks along edge paths on the combinatorial graph structure defined by the boundary of convex polyhedra. Since these graphs are quite dense (Balinski's theorem states that the graph of d-dimensional polyhedron must be d-connected [Ziegler 1995]) and possibly large (the Lower Bound Theorem states that the number of vertices can be exponential in the dimension [Ziegler 1995]), it is indeed somewhat of a miracle that it manages to get to an optimal extreme point as quickly as it does. Empirical and probabilistic analyses indicate that the number of iterations of the simplex method is just slightly more than linear in the dimension of the primal polyhedron. However, there is no known variant of the simplex method with a worst-case polynomial guarantee on the number of iterations. Even a polynomial bound on the diameter of polyhedral graphs is not known.

Procedure 15.1 Primal Simplex ($\mathcal{K}$, $\boldsymbol{c}$):

0. **Initialize:**

   $\mathbf{x}_0 :=$ an extreme point of $\mathcal{K}$
   $k := 0$

1. **Iterative step:**

   do
   If for all edge directions $\mathcal{D}_k$ at $\mathbf{x}_k$, the objective function is nondecreasing, i.e.,

$$\mathbf{cd} \geq 0 \quad \forall\, \mathbf{d} \in \mathcal{D}_k$$

   then exit and return optimal $\mathbf{x}_k$.
   Else pick some $\mathbf{d}_k$ in $\mathcal{D}_k$ such that $\mathbf{cd}_k < 0$.
   If $\mathbf{d}_k \geq 0$ **then** declare the linear program unbounded in objective value and exit.
   Else $\mathbf{x}_{k+1} := \mathbf{x}_k + \theta_k * \mathbf{d}_k$, where

$$\theta_k = \max\{\theta : \mathbf{x}_k + \theta * \mathbf{d}_k \geq 0\}$$

   $k := k + 1$
   od

2. **End**

Remark 15.1 In the initialization step, we assumed that an extreme point $\mathbf{x}_0$ of the polyhedron $\mathcal{K}$ is available. This also assumes that the solvability of the constraints defining $\mathcal{K}$ has been established. These

assumptions are reasonable since we can formulate the solvability problem as an optimization problem, with a self-evident extreme point, whose optimal solution either establishes unsolvability of $A\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$ or provides an extreme point of $\mathcal{K}$. Such an optimization problem is usually called a phase I model. The point being, of course, that the simplex method, as just described, can be invoked on the phase I model and, if successful, can be invoked once again to carry out the intended minimization of $\mathbf{cx}$. There are several different formulations of the phase I model that have been advocated. Here is one:

$$\min\{v_0 \ : \ A\mathbf{x} + \mathbf{b}v_0 = \mathbf{b},\ \mathbf{x} \geq \mathbf{0},\ v_0 \geq 0\}$$

The solution $(\mathbf{x}, v_0)^T = (0, \ldots, 0, 1)$ is a self-evident extreme point and $v_0 = 0$ at an optimal solution of this model is a necessary and sufficient condition for the solvability of $A\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$.

Remark 15.2 The scheme for generating improving edge directions uses an algebraic representation of the extreme points as certain bases, called feasible bases, of the vector space generated by the columns of the matrix A. It is possible to have linear programs for which an extreme point is geometrically overdetermined (degenerate), i.e., there are more than d facets of $\mathcal{K}$ that contain the extreme point, where d is the dimension of $\mathcal{K}$. In such a situation, there would be several feasible bases corresponding to the same extreme point. When this happens, the linear program is said to be *primal degenerate*.

Remark 15.3 There are two sources of nondeterminism in the primal simplex procedure. The first involves the choice of edge direction $\mathbf{d}_k$ made in step 1. At a typical iteration there may be many edge directions that are improving in the sense that $\mathbf{cd}_k < 0$. Dantzig's rule, the maximum improvement rule, and steepest descent rule are some of the many rules that have been used to make the choice of edge direction in the simplex method. There is, unfortunately, no clearly dominant rule and successful codes exploit the empirical and analytic insights that have been gained over the years to resolve the edge selection nondeterminism in simplex methods. The second source of nondeterminism arises from degeneracy. When there are multiple feasible bases corresponding to an extreme point, the simplex method has to pivot from basis to adjacent basis by picking an entering basic variable (a pseudoEdge direction) and by dropping one of the old ones. A wrong choice of the leaving variables may lead to cycling in the sequence of feasible bases generated at this extreme point. Cycling is a serious problem when linear programs are highly degenerate as in the case of linear relaxations of many combinatorial optimization problems. The lexicographic rule (perturbation rule) for the choice of leaving variables in the simplex method is a provably finite method (i.e., all cycles are broken). A clever method proposed by Bland (cf. Schrijver [1986]) preorders the rows and columns of the matrix A. In the case of nondeterminism in either entering or leaving variable choices, Bland's rule just picks the lowest index candidate. All cycles are avoided by this rule also.

The simplex method has been the veritable workhorse of linear programming for four decades now. However, as already noted, we do not know of a simplex method that has worst-case bounds that are polynomial. In fact, Klee and Minty exploited the sensitivity of the original simplex method of Dantzig, to projective scaling of the data, and constructed exponential examples for it. Recently, Spielman and Tang [2001] introduced the concept of smoothed analysis and smoothed complexity of algorithms, which is a hybrid of worst-case and average-case analysis of algorithms. Essentially, this involves the study of performance of algorithms under small random Gaussian perturbations of the coefficients of the constraint matrix. The authors show that a variant of the simplex algorithm, known as the *shadow vertex simplex algorithm* (Gass and Saaty [1955]) has polynomial smoothed complexity.

The ellipsoid method of Shor [1970] was devised to overcome poor scaling in convex programming problems and, therefore, turned out to be the natural choice of an algorithm to first establish polynomial-time solvability of linear programming. Later Karmarkar [1984] took care of both projection and scaling simultaneously and arrived at a superior algorithm.

15.2.1.3 The Ellipsoid Algorithm

The ellipsoid algorithm of Shor [1970] gained prominence in the late 1970s when Haĉijan [1979] (pronounced Khachiyan) showed that this convex programming method specializes to a polynomial-time algorithm for linear programming problems. This theoretical breakthrough naturally led to intense study of this method and its properties. The survey paper by Bland et al. [1981] and the monograph by Akgül [1984] attest to this fact. The direct theoretical consequences for combinatorial optimization problems was independently documented by Padberg and Rao [1981], Karp and Papadimitriou [1982], and Grötschel et al. [1988]. The ability of this method to implicitly handle linear programs with an exponential list of constraints and maintain polynomial-time convergence is a characteristic that is the key to its applications in combinatorial optimization. For an elegant treatment of the many deep theoretical consequences of the ellipsoid algorithm, the reader is directed to the monograph by Lovász [1986] and the book by Grötschel et al. [1988].

Computational experience with the ellipsoid algorithm, however, showed a disappointing gap between the theoretical promise and practical efficiency of this method in the solution of linear programming problems. Dense matrix computations as well as the slow average-case convergence properties are the reasons most often cited for this behavior of the ellipsoid algorithm. On the positive side though, it has been noted (cf. Ecker and Kupferschmid [1983]) that the ellipsoid method is competitive with the best known algorithms for (nonlinear) convex programming problems.

Let us consider the problem of testing if a polyhedron $\mathcal{Q} \in \Re^d$, defined by linear inequalities, is nonempty. For technical reasons let us assume that $\mathcal{Q}$ is rational, i.e., all extreme points and rays of $\mathcal{Q}$ are rational vectors or, equivalently, that all inequalities in some description of $\mathcal{Q}$ involve only rational coefficients. The ellipsoid method does not require the linear inequalities describing $\mathcal{Q}$ to be explicitly specified. It suffices to have an oracle representation of $\mathcal{Q}$. Several different types of oracles can be used in conjunction with the ellipsoid method (Karp and Papadimitriou [1982], Padberg and Rao [1981], Grötschel et al. [1988]). We will use the *strong separation oracle:*

Oracle: **Strong Separation**($\mathcal{Q}$, $\mathbf{y}$)

> Given a vector $\mathbf{y} \in \Re^d$, decide whether $\mathbf{y} \in \mathcal{Q}$, and if not find a hyperplane that separates $\mathbf{y}$ from $\mathcal{Q}$; more precisely, find a vector $\mathbf{c} \in \Re^d$ such that $\mathbf{c}^T\mathbf{y} < \min\{\mathbf{c}^T\mathbf{x} \mid \mathbf{x} \in \mathcal{Q} \mid$.

The ellipsoid algorithm initially chooses an ellipsoid large enough to contain a part of the polyhedron $\mathcal{Q}$ if it is nonempty. This is easily accomplished because we know that if $\mathcal{Q}$ is nonempty then it has a rational solution whose (binary encoding) length is bounded by a polynomial function of the length of the largest coefficient in the linear program and the dimension of the space.

The center of the ellipsoid is a feasible point if the separation oracle tells us so. In this case, the algorithm terminates with the coordinates of the center as a solution. Otherwise, the separation oracle outputs an inequality that separates the center point of the ellipsoid from the polyhedron $\mathcal{Q}$. We translate the hyperplane defined by this inequality to the center point. The hyperplane slices the ellipsoid into two halves, one of which can be discarded. The algorithm now creates a new ellipsoid that is the minimum volume ellipsoid containing the remaining half of the old one. The algorithm questions if the new center is feasible and so on. The key is that the new ellipsoid has substantially smaller volume than the previous one. When the volume of the current ellipsoid shrinks to a sufficiently small value, we are able to conclude that $\mathcal{Q}$ is empty. This fact is used to show the polynomial-time convergence of the algorithm.

The crux of the complexity analysis of the algorithm is on the a priori determination of the iteration bound. This in turn depends on three factors. The volume of the initial ellipsoid E_0, the rate of volume shrinkage $(vol(E_{k+1})/vol(E_k) < e^{-\frac{1}{(2d)}})$, and the volume threshold at which we can safely conclude that $\mathcal{Q}$ must be empty. The assumption of $\mathcal{Q}$ being a rational polyhedron is used to argue that $\mathcal{Q}$ can be

modified into a full-dimensional polytope without affecting the decision question: "Is $\mathcal{Q}$ non-empty?" After careful accounting for all of these technical details and some others (e.g., compensating for the roundoff errors caused by the square root computation in the algorithm), it is possible to establish the following fundamental result.

Theorem 15.1 *There exists a polynomial* $g(d, \phi)$ *such that the* **ellipsoid method** *runs in time bounded by* $T\, g(d, \phi)$ *where* ϕ *is an upper bound on the size of linear inequalities in some description of* $\mathcal{Q}$ *and* T *is the maximum time required by the oracle* **Strong Separation**$(\mathcal{Q}, \mathbf{y})$ *on inputs* $\mathbf{y}$ *of size at most* $g(d, \phi)$.

The size of a linear inequality is just the length of the encoding of all of the coefficients needed to describe the inequality. A direct implication of the theorem is that solvability of linear inequalities can be checked in polynomial time if strong separation can be solved in polynomial time. This implies that the standard linear programming solvability question has a polynomial-time algorithm (since separation can be effected by simply checking all of the constraints). Happily, this approach provides polynomial-time algorithms for much more than just the standard case of linear programming solvability. The theorem can be extended to show that the optimization of a linear objective function over $\mathcal{Q}$ also reduces to a polynomial number of calls to the strong separation oracle on $\mathcal{Q}$. A converse to this theorem also holds, namely, separation can be solved by a polynomial number of calls to a solvability/optimization oracle (Grötschel et al. [1982]). Thus, optimization and separation are polynomially equivalent. This provides a very powerful technique for identifying tractable classes of optimization problems. Semidefinite programming and submodular function minimization are two important classes of optimization problems that can be solved in polynomial time using this property of the ellipsoid method.

15.2.1.4 Semidefinite Programming

The following optimization problem defined on symmetric $(n \times n)$ real matrices

$$\text{(SDP)} \qquad \min_{X \in \Re^{n \times n}} \left\{ \sum_{ij} C \bullet X : A \bullet X = B,\ X \succeq 0 \right\}$$

is called a semidefinite program. Note that $X \succeq 0$ denotes the requirement that X is a positive semidefinite matrix, and $F \bullet G$ for $n \times n$ matrices F and G denotes the product matrix $(F_{ij} * G_{ij})$. From the definition of positive semidefinite matrices, $X \succeq 0$ is equivalent to

$$\mathbf{q}^T X \mathbf{q} \geq 0 \quad \text{for every } \mathbf{q} \in \Re^n$$

Thus semidefinite programming (SDP) is really a linear program on $O(n^2)$ variables with an (uncountably) infinite number of linear inequality constraints. Fortunately, the strong separation oracle is easily realized for these constraints. For a given symmetric X we use Cholesky factorization to identify the minimum eigenvalue $\lambda_{\min}$. If $\lambda_{\min}$ is nonnegative then $X \succeq 0$ and if, on the other hand, $\lambda_{\min}$ is negative we have a separating inequality

$$\boldsymbol{\gamma}_{\min}^T X \boldsymbol{\gamma}_{\min} \geq 0$$

where $\boldsymbol{\gamma}_{\min}$ is the eigenvector corresponding to $\lambda_{\min}$. Since the Cholesky factorization can be computed by an $O(n^3)$ algorithm, we have a polynomial-time separation oracle and an efficient algorithm for SDP via the ellipsoid method. Alizadeh [1995] has shown that interior point methods can also be adapted to solving SDP to within an additive error ϵ in time polynomial in the size of the input and $\log 1/\epsilon$.

This result has been used to construct efficient approximation algorithms for maximum stable sets and cuts of graphs, Shannon capacity of graphs, and minimum colorings of graphs. It has been used to define hierarchies of relaxations for integer linear programs that strictly improve on known exponential-size linear programming relaxations. We shall encounter the use of SDP in the approximation of a maximum weight cut of a given vertex-weighted graph in Section 15.7.

15.2.1.5 Minimizing Submodular Set Functions

The minimization of submodular set functions is another important class of optimization problems for which ellipsoidal and projective scaling algorithms provide polynomial-time solution methods.

Definition 15.1 Let N be a finite set. A real valued set function f defined on the subsets of N is submodular if $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$ for $X, Y \subseteq N$.

Example 15.1

Let $G = (V, E)$ be an undirected graph with V as the node set and E as the edge set. Let $c_{ij} \geq 0$ be the weight or capacity associated with edge $(ij) \in E$. For $S \subseteq V$, define the cut function $c(S) = \sum_{i \in S,\, j \in V \setminus S} c_{ij}$. The cut function defined on the subsets of V is submodular since $c(X) + c(Y) - c(X \cup Y) - c(X \cap Y) = \sum_{i \in X \setminus Y,\, j \in Y \setminus X} 2c_{ij} \geq 0$.

The optimization problem of interest is

$$\min\{f(X) : X \subseteq N\}$$

The following remarkable construction that connects submodular function minimization with convex function minimization is due to Lovász (see Grötschel et al. [1988]).

Definition 15.2 The Lovász extension $\hat{f}(.)$ of a submodular function $f(.)$ satisfies

- $\hat{f} : [0, 1]^N \to \Re$.
- $\hat{f}(\mathbf{x}) = \sum_{I \in \mathcal{I}} \lambda_I f(\mathbf{x}_I)$ where $\mathbf{x} = \sum_{I \in \mathcal{I}} \lambda_I \mathbf{x}_I$, $\mathbf{x} \in [0, 1]^N$, $\mathbf{x}_I$ is the incidence vector of I for each $I \in \mathcal{I}$, $\lambda_I > 0$ for each I in $\mathcal{I}$, and $\mathcal{I} = \{I_1, I_2, \ldots, I_k\}$ with $\emptyset \neq I_1 \subset I_2 \subset \cdots \subset I_k \subseteq N$. Note that the representation $\mathbf{x} = \sum_{I \in \mathcal{I}} \lambda_I \mathbf{x}_I$ is unique given that the $\lambda_I > 0$ and that the sets in $\mathcal{I}$ are nested.

It is easy to check that $\hat{f}(.)$ is a convex function. Lovász also showed that the minimization of the submodular function $f(.)$ is a special case of convex programming by proving

$$\min\{f(X) : X \subseteq N\} = \min\{\hat{f}(\mathbf{x}) : \mathbf{x} \in [0, 1]^N\}$$

Further, if $\mathbf{x}^*$ is an optimal solution to the convex program and

$$\mathbf{x}^* = \sum_{I \in \mathcal{I}} \lambda_I \mathbf{x}_I$$

then for each $\lambda_I > 0$, it can be shown that $I \in \mathcal{I}$ minimizes f. The ellipsoid method can be used to solve this convex program (and hence submodular minimization) using a polynomial number of calls to an oracle for f [this oracle returns the value of $f(X)$ when input X].

15.2.1.6 Interior Point Methods

The announcement of the polynomial solvability of linear programming followed by the probabilistic analyses of the simplex method in the early 1980s left researchers in linear programming with a dilemma. We had one method that was good in a theoretical sense but poor in practice and another that was good in practice (and on average) but poor in a theoretical worst-case sense. This left the door wide open for a method that was good in both senses. Narendra Karmarkar closed this gap with a breathtaking new projective scaling algorithm. In retrospect, the new algorithm has been identified with a class of nonlinear programming methods known as logarithmic barrier methods. Implementations of a primal–dual variant of the logarithmic barrier method have proven to be the best approach at present. It is this variant that we describe.

It is well known that moving through the interior of the feasible region of a linear program using the negative of the gradient of the objective function, as the movement direction, runs into trouble because of getting *jammed* into corners (in high dimensions, corners make up most of the interior of a polyhedron). This jamming can be overcome if the negative gradient is balanced with a *centering* direction. The centering

direction in Karmarkar's algorithm is based on the *analytic center* $\mathbf{y}_c$ of a full-dimensional polyhedron $\mathcal{D} = \{\mathbf{y} : A^T\mathbf{y} \leq c\}$ which is the unique optimal solution to

$$\max\left\{\sum_{j=1}^{n} \ell n\,(\mathbf{z}_j)\ :\ A^T\mathbf{y} + \mathbf{z} = \mathbf{c}\right\}$$

Recall the primal and dual forms of a linear program may be taken as

$$\begin{aligned} &(P) \quad \min\{\mathbf{cx} : A\mathbf{x} = \mathbf{b},\, \mathbf{x} \geq \mathbf{0}\} \\ &(D) \quad \max\{\mathbf{b}^T\mathbf{y} : A^T\mathbf{y} \leq \mathbf{c}\} \end{aligned}$$

The logarithmic barrier formulation of the dual (D) is

$$(D_\mu) \quad \max\left\{\mathbf{b}^T\mathbf{y} + \mu\sum_{j=1}^{n} \ell n\,(\mathbf{z}_j) :\ A^T\mathbf{y} + \mathbf{z} = \mathbf{c}\right\}$$

Notice that (D_μ) is equivalent to (D) as $\mu \to 0^+$. The optimality (Karush–Kuhn–Tucker) conditions for (D_μ) are given by

$$\begin{aligned} D_\mathbf{x} D_\mathbf{z}\mathbf{e} &= \mu\mathbf{e} \\ A\mathbf{x} &= \mathbf{b} \\ A^T\mathbf{y} + \mathbf{z} &= \mathbf{c} \end{aligned} \tag{15.1}$$

where $D_\mathbf{x}$ and $D_\mathbf{z}$ denote $n \times n$ diagonal matrices whose diagonals are $\mathbf{x}$ and $\mathbf{z}$, respectively. Notice that if we set μ to 0, the above conditions are precisely the primal–dual optimality conditions: complementary slackness, primal and dual feasibility of a pair of optimal (P) and (D) solutions. The problem has been reduced to solving the equations in $\mathbf{x}, \mathbf{y}, \mathbf{z}$. The classical technique for solving equations is Newton's method, which prescribes the directions,

$$\begin{aligned} \Delta\mathbf{y} &= -\left(AD_\mathbf{x}D_\mathbf{z}^{-1}A^T\right)^{-1} AD_\mathbf{z}^{-1}(\mu\mathbf{e} - D_\mathbf{x}D_\mathbf{z}\mathbf{e})\Delta\mathbf{z} = -A^T\Delta\mathbf{y}\Delta\mathbf{x} \\ &= D_\mathbf{z}^{-1}(\mu\mathbf{e} - D_\mathbf{x}D_\mathbf{z}\mathbf{e}) - D_\mathbf{x}D_\mathbf{z}^{-1}\Delta\mathbf{z} \end{aligned} \tag{15.2}$$

The strategy is to take one Newton step, reduce μ, and iterate until the optimization is complete. The criterion for stopping can be determined by checking for feasibility ($\mathbf{x}, \mathbf{z} \geq \mathbf{0}$) and if the duality gap ($\mathbf{x}^t\mathbf{z}$) is close enough to 0. We are now ready to describe the algorithm.

Procedure 15.2 Primal-Dual Interior:

0. **Initialize:**

   $\mathbf{x}_0 > 0,\ \mathbf{y}_0 \in \Re^m,\ \mathbf{z}_0 > 0,\ \mu_0 > 0,\ \epsilon > 0,\ \rho > 0$
   $k := 0$

1. **Iterative step:**

   do
   Stop if $A\mathbf{x}_k = \mathbf{b}$, $A^T\mathbf{y}_k + \mathbf{z}_k = \mathbf{c}$ and $\mathbf{x}_k^T\mathbf{z}_k \leq \epsilon$.
   $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k^P\Delta\mathbf{x}_k$
   $\mathbf{y}_{k+1} \leftarrow \mathbf{y}_k + \alpha_k^D\Delta\mathbf{y}_k$
   $\mathbf{z}_{k+1} \leftarrow \mathbf{z}_k + \alpha_k^D\Delta\mathbf{z}_k$
   /* $\Delta\mathbf{x}_k$, $\Delta\mathbf{y}_k$, $\Delta\mathbf{z}_k$ **are the Newton directions from (1)** */
   $\mu_{k+1} \leftarrow \rho\mu_k$
   $k := k + 1$
   od

2. **End**

Remark 15.4 The step sizes α_k^P and α_k^D are chosen to keep $\mathbf{x}_{k+1}$ and $\mathbf{z}_{k+1}$ strictly positive. The ability in the primal–dual scheme to choose separate step sizes for the primal and dual variables is a major advantage that this method has over the pure primal or dual methods. Empirically this advantage translates to a significant reduction in the number of iterations.

Remark 15.5 The stopping condition essentially checks for primal and dual feasibility and near complementary slackness. Exact complementary slackness is not possible with interior solutions. It is possible to maintain primal and dual feasibility through the algorithm, but this would require a phase I construction via artificial variables. Empirically, this feasible variant has not been found to be worthwhile. In any case, when the algorithm terminates with an interior solution, a post-processing step is usually invoked to obtain optimal extreme point solutions for the primal and dual. This is usually called the *purification* of solutions and is based on a clever scheme described by Megiddo [1991].

Remark 15.6 Instead of using Newton steps to drive the solutions to satisfy the optimality conditions of (D_μ), Mehrotra [1992] suggested a predictor–corrector approach based on power series approximations. This approach has the added advantage of providing a rational scheme for reducing the value of μ. It is the predictor–corrector based primal–dual interior method that is considered the current winner in interior point methods. The OB1 code of Lustig et al. [1994] is based on this scheme.

Remark 15.7 CPLEX 6.5 [1999], a general purpose linear (and integer) programming solver, contains implementations of interior point methods. A computational study of parallel implementations of simplex and interior point methods on the SGI power challenge (SGI R8000) platform indicates that on all but a few small linear programs in the NETLIB linear programming benchmark problem set, interior point methods dominate the simplex method in run times. New advances in handling Cholesky factorizations in parallel are apparently the reason for this exceptional performance of interior point methods. For the simplex method, CPLEX 6.5 incorporates efficient methods of solving triangular linear systems and faster updating of reduced costs for identifying improving edge directions. For the interior point method, the same code includes improvements in computing Cholesky factorizations and better use of level-two cache available in modern computing architectures. Using CPLEX 6.5 and CPLEX 5.0, Bixby et al. [2001] in a recent study have done extensive computational testing comparing the two codes with respect to the performance of the Primal simplex, Dual simplex and Interior Point methods as well as a comparison of the performance of these three methods. While CPLEX 6.5 considerably outperformed CPLEX 5.0 for all the three methods, the comparison among the three methods is inconclusive. However, as stated by Bixby et al. [2001], the computational testing was biased against interior point method because of the inferior floating point performance of the machine used and the nonimplementation of the parallel features on shared memory machines.

Remark 15.8 Karmarkar [1990] has proposed an interior-point approach for integer programming problems. The main idea is to reformulate an integer program as the minimization of a quadratic energy function over linear constraints on continuous variables. Interior-point methods are applied to this formulation to find local optima.

15.3 Large-Scale Linear Programming in Combinatorial Optimization

Linear programming problems with thousands of rows and columns are routinely solved either by variants of the simplex method or by interior point methods. However, for several linear programs that arise in combinatorial optimization, the number of columns (or rows in the dual) are too numerous to be enumerated explicitly. The columns, however, often have a structure which is exploited to generate the columns as and when required in the simplex method. Such an approach, which is referred to as **column**

generation, is illustrated next on the *cutting stock problem* (Gilmore and Gomory [1963]), which is also known as the *bin packing problem* in the computer science literature.

15.3.1 Cutting Stock Problem

Rolls of sheet metal of standard length L are used to cut required lengths $l_i, i = 1, 2, \ldots, m$. The jth cutting pattern should be such that a_{ij}, the number of sheets of length l_i cut from one roll of standard length L, must satisfy $\sum_{i=1}^{m} a_{ij} l_i \leq L$. Suppose $n_i, i = 1, 2, \ldots, m$ sheets of length l_i are required. The problem is to find cutting patterns so as to minimize the number of rolls of standard length L that are used to meet the requirements. A linear programming formulation of the problem is as follows.

Let $\mathbf{x}_j$, $j = 1, 2, \ldots, n$, denote the number of times the jth cutting pattern is used. In general, $\mathbf{x}_j$, $j = 1, 2, \ldots, n$ should be an integer but in the next formulation the variables are permitted to be fractional.

$$\text{(P1)} \quad \text{Min} \sum_{j=1}^{n} \mathbf{x}_j$$

$$\begin{aligned} \text{Subject to} \quad & \sum_{j=1}^{n} a_{ij} \mathbf{x}_j \geq n_i & i = 1, 2, \ldots, m \\ & \mathbf{x}_j \geq 0 & j = 1, 2, \ldots, n \\ \text{where} \quad & \sum_{i=1}^{m} l_i a_{ij} \leq L & j = 1, 2, \ldots, n \end{aligned}$$

The formulation can easily be extended to allow for the possibility of p standard lengths L_k, $k = 1, 2, \ldots, p$, from which the n_i units of length l_i, $i = 1, 2, \ldots, m$, are to be cut.

The cutting stock problem can also be viewed as a bin packing problem. Several bins, each of standard capacity L, are to be packed with n_i units of item i, each of which uses up capacity of l_i in a bin. The problem is to minimize the number of bins used.

15.3.1.1 Column Generation

In general, the number of columns in (P1) is too large to enumerate all of the columns explicitly. The simplex method, however, does not require all of the columns to be explicitly written down. Given a basic feasible solution and the corresponding simplex multipliers $\mathbf{w}_i, i = 1, 2, \ldots, m$, the column to enter the basis is determined by applying dynamic programming to solve the following knapsack problem:

$$\text{(P2)} \quad z = \text{Max} \sum_{i=1}^{m} \mathbf{w}_i a_i$$

$$\begin{aligned} \text{Subject to} \quad & \sum_{i=1}^{m} l_i a_i \leq L \\ & a_i \geq 0 \text{ and integer,} \quad \text{for } i = 1, 2, \ldots, m \end{aligned}$$

Let a_i^*, $i = 1, 2, \ldots, m$, denote an optimal solution to (P2). If $z > 1$, the kth column to enter the basis has coefficients $a_{ik} = a_i^*$, $i = 1, 2, \ldots, m$.

Using the identified columns, a new improved (in terms of the objective function value) basis is obtained, and the column generation procedure is repeated. A major iteration is one in which (P2) is solved to identify, if there is one, a column to enter the basis. Between two major iterations, several minor iterations may be performed to optimize the linear program using only the available (generated) columns.

If $z \leq 1$, the current basic feasible solution is optimal to (P1). From a computational point of view, alternative strategies are possible. For instance, instead of solving (P2) to optimality, a column to enter the basis can be indentified as soon as a feasible solution to (P2) with an objective function value greater than 1 has been found. Such an approach would reduce the time required to solve (P2) but may increase the number of iterations required to solve (P1).

A column once generated may be retained, even if it comes out of the basis at a subsequent iteration, so as to avoid generating the same column again later on. However, at a particular iteration some columns, which appear unattractive in terms of their reduced costs, may be discarded in order to avoid having to store a large number of columns. Such columns can always be generated again subsequently, if necessary. The rationale for this approach is that such unattractive columns will rarely be required subsequently.

The dual of (P1) has a large number of rows. Hence column generation may be viewed as row generation in the dual. In other words, in the dual we start with only a few constraints explicitly written down. Given an optimal solution $\mathbf{w}$ to the current dual problem (i.e., with only a few constraints which have been explicitly written down) find a constraint that is violated by $\mathbf{w}$ or conclude that no such constraint exists. The problem to be solved for identifying a violated constraint, if any, is exactly the separation problem that we encountered in the section on algorithms for linear programming.

15.3.2 Decomposition and Compact Representations

Large-scale linear programs sometimes have a block diagonal structure with a few additional constraints linking the different blocks. The linking constraints are referred to as the master constraints and the various blocks of constraints are referred to as subproblem constraints. Using the representation theorem of polyhedra (see, for instance, Nemhauser and Wolsey [1988]), the decomposition approach of Dantzig and Wolfe [1961] is to convert the original problem to an equivalent linear program with a small number of constraints but with a large number of columns or variables. In the cutting stock problem described in the preceding section, the columns are generated, as and when required, by solving a knapsack problem via dynamic programming. In the Dantzig–Wolfe decomposition scheme, the columns are generated, as and when required, by solving appropriate linear programs on the subproblem constraints.

It is interesting to note that the reverse of decomposition is also possible. In other words, suppose we start with a statement of a problem and an associated linear programming formulation with a large number of columns (or rows in the dual). If the column generation (or row generation in the dual) can be accomplished by solving a linear program, then a *compact* formulation of the original problem can be obtained. Here compact refers to the number of rows and columns being bounded by a polynomial function of the input length of the original problem. This result due to Martin [1991] enables one to solve the problem in the polynomial time by solving the compact formulation using interior point methods.

15.4 Integer Linear Programs

Integer linear programming problems (ILPs) are linear programs in which all of the variables are restricted to be integers. If only some but not all variables are restricted to be integers, the problem is referred to as a mixed integer program. Many combinatorial problems can be formulated as integer linear programs in which all of the variables are restricted to be 0 or 1. We will first discuss several examples of combinatorial optimization problems and their formulation as integer programs. Then we will review a general representation theory for integer programs that gives a formal measure of the expressiveness of this algebraic approach. We conclude this section with a representation theorem due to Benders [1962], which has been very useful in solving certain large-scale combinatorial optimization problems in practice.

15.4.1 Example Formulations

15.4.1.1 Covering and Packing Problems

A wide variety of location and scheduling problems can be formulated as set covering or set packing or set partitioning problems. The three different types of **covering and packing** problems can be succinctly stated as follows: Given (1) a finite set of elements $\mathcal{M} = \{1, 2, \ldots, m\}$, and (2) a family F of subsets of $\mathcal{M}$ with each member $F_j, j = 1, 2, \ldots, n$ having a profit (or cost) $\mathbf{c}_j$ associated with it, find a collection, S,

of the members of F that maximizes the profit (or minimizes the cost) while ensuring that every element of $\mathcal{M}$ is in one of the following:

(P3): at most one member of S (set packing problem)
(P4): at least one member of S (set covering problem)
(P5): exactly one member of S (set partitioning problem)

The three problems (P3), (P4), and (P5) can be formulated as ILPs as follows:

Let A denote the $m \times n$ matrix where

$$A_{ij} = \begin{cases} 1 & \text{if element } i \in F_j \\ 0 & \text{otherwise} \end{cases}$$

The decision variables are $\mathbf{x}_j$, $j = 1, 2, \ldots, n$ where

$$\mathbf{x}_{ij} = \begin{cases} 1 & \text{if } F_j \text{ is chosen} \\ 0 & \text{otherwise} \end{cases}$$

The set packing problem is

$$\begin{aligned} \text{(P3)} \quad & \text{Max } \mathbf{cx} \\ \text{Subject to} \quad & A\mathbf{x} \leq \mathbf{e}_m \\ & \mathbf{x}_j = 0 \quad \text{or} \quad 1, \quad j = 1, 2, \ldots, n \end{aligned}$$

where $\mathbf{e}_m$ is an m-dimensional column vector of ones.

The set covering problem (P4) is (P3) with less than or equal to constraints replaced by greater than or equal to constraints and the objective is to minimize rather than maximize. The set partitioning problem (P5) is (P3) with the constraints written as equalities. The set partitioning problem can be converted to a set packing problem or set covering problem (see Padberg [1995]) using standard transformations. If the right-hand side vector $\mathbf{e}_m$ is replaced by a nonnegative integer vector $\mathbf{b}$, (P3) is referred to as the generalized set packing problem.

The airline crew scheduling problem is a classic example of the set partitioning or the set covering problem. Each element of $\mathcal{M}$ corresponds to a flight segment. Each subset F_j corresponds to an acceptable set of flight segments of a crew. The problem is to cover, at minimum cost, each flight segment exactly once. This is a set partitioning problem. If *dead heading* of crew is permitted, we have the set covering problem.

15.4.1.2 Packing and Covering Problems in a Graph

Suppose A is the node-edge incidence matrix of a graph. Now, (P3) is a weighted matching problem. If in addition, the right-hand side vector $\mathbf{e}_m$ is replaced by a nonnegative integer vector $\mathbf{b}$, (P3) is referred to as a weighted $\mathbf{b}$-matching problem. In this case, each variable $\mathbf{x}_j$ which is restricted to be an integer may have a positive upper bound of u_j. Problem (P4) is now referred to as the weighted edge covering problem. Note that by substituting for $\mathbf{x}_j = 1 - \mathbf{y}_j$, where $\mathbf{y}_j = 0$ or 1, the weighted edge covering problem is transformed to a weighted $\mathbf{b}$-matching problem in which the variables are restricted to be 0 or 1.

Suppose A is the edge-node incidence matrix of a graph. Now, (P3) is referred to as the weighted vertex packing problem and (P4) is referred to as the weighted vertex covering problem. The *set packing* problem can be transformed to a weighted vertex packing problem in a graph G as follows:

> G contains a node for each $\mathbf{x}_j$ and an edge between nodes j and k exists if and only if the columns $A_{.j}$ and $A_{.k}$ are not orthogonal. G is called the *intersection graph* of A. The set packing problem is equivalent to the weighted vertex packing problem on G. Given G, the complement graph $\overline{G}$ has the same node set as G and there is an edge between nodes j and k in $\overline{G}$ if and only if there is no such corresponding edge in G. A clique in a graph is a subset, k, of nodes of G such that the subgraph induced by k is complete. Clearly, the weighted vertex packing problem in G is equivalent to finding a maximum weighted clique in $\overline{G}$.

15.4.1.3 Plant Location Problems

Given a set of customer locations $N = \{1, 2, \dots, n\}$ and a set of potential sites for plants $M = \{1, 2, \dots, m\}$, the plant location problem is to identify the sites where the plants are to be located so that the customers are served at a minimum cost. There is a fixed cost $\mathbf{f}_i$ of locating the plant at site i and the cost of serving customer j from site i is $\mathbf{c}_{ij}$. The decision variables are: $\mathbf{y}_i$ is set to 1 if a plant is located at site i and to 0 otherwise; $\mathbf{x}_{ij}$ is set to 1 if site i serves customer j and to 0 otherwise.

A formulation of the problem is

$$\text{(P6)} \quad \text{Min} \sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{c}_{ij}\mathbf{x}_{ij} + \sum_{i=1}^{m} \mathbf{f}_i \mathbf{y}_i$$

$$\begin{aligned} \text{subject to} \quad \sum_{i=1}^{m} \mathbf{x}_{ij} &= 1 && j = 1, 2, \dots, n \\ \mathbf{x}_{ij} - \mathbf{y}_i &\le 0 && i = 1, 2, \dots, m; \quad j = 1, 2, \dots, n \\ \mathbf{y}_i &= 0 \quad \text{or} \quad 1 && i = 1, 2, \dots, m \\ \mathbf{x}_{ij} &= 0 \quad \text{or} \quad 1 && i = 1, 2, \dots, m; \quad j = 1, 2, \dots, n \end{aligned}$$

Note that the constraints $\mathbf{x}_{ij} - \mathbf{y}_i \le 0$ are required to ensure that customer j may be served from site i only if a plant is located at site i. Note that the constraints $\mathbf{y}_i = 0$ or 1 force an optimal solution in which $\mathbf{x}_{ij} =$ 0 or 1. Consequently, the $\mathbf{x}_{ij} = 0$ or 1 constraints may be replaced by nonnegativity constraints $\mathbf{x}_{ij} \ge 0$.

The linear programming relaxation associated with (P6) is obtained by replacing constraints $\mathbf{y}_i = 0$ or 1 and $\mathbf{x}_{ij} = 0$ or 1 by nonnegativity contraints on $\mathbf{x}_{ij}$ and $\mathbf{y}_i$. The upper bound constraints on $\mathbf{y}_i$ are not required provided $\mathbf{f}_i \ge 0, i = 1, 2, \dots, m$. The upper bound constraints on $\mathbf{x}_{ij}$ are not required in view of constraints $\sum_{i=1}^{m} \mathbf{x}_{ij} = 1$.

Remark 15.9 It is frequently possible to formulate the same combinatorial problem as two or more different ILPs. Suppose we have two ILP formulations (F1) and (F2) of the given combinatorial problem with both (F1) and (F2) being minimizing problems. Formulation (F1) is said to be stronger than (F2) if (LP1), the the linear programming relaxation of (F1), always has an optimal objective function value which is greater than or equal to the optimal objective function value of (LP2), which is the linear programming relaxation of (F2).

It is possible to reduce the number of constraints in (P6) by replacing the constraints $\mathbf{x}_{ij} - \mathbf{y}_i \le 0$ by an aggregate:

$$\sum_{j=1}^{n} \mathbf{x}_{ij} - n\mathbf{y}_i \le 0 \quad i = 1, 2, \dots, m$$

However, the disaggregated (P6) is a stronger formulation than the formulation obtained by aggregrating the constraints as previously. By using standard transformations, (P6) can also be converted into a set packing problem.

15.4.1.4 Satisfiability and Inference Problems:

In propositional logic, a truth assignment is an assignment of true or false to each atomic proposition $\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_n$. A literal is an atomic proposition $\mathbf{x}_j$ or its negation $\neg\mathbf{x}_j$. For propositions in conjunctive normal form, a clause is a disjunction of literals and the proposition is a conjunction of clauses. A clause is obviously satisfied by a given truth assignment if at least one of its literals is true. The satisfiability problem consists of determining whether there exists a truth assignment to atomic propositions such that a set S of clauses is satisfied.

Let T_i denote the set of atomic propositions such that if any one of them is assigned true, the clause $i \in S$ is satisfied. Similarly, let F_i denote the set of atomic propositions such that if any one of them is assigned false, the clause $i \in S$ is satisfied.

The decision variables are

$$\mathbf{x}_j = \begin{cases} 1 & \text{if atomic proposition } j \text{ is assigned true} \\ 0 & \text{if atomic proposition } j \text{ is assigned false} \end{cases}$$

The satisfiability problem is to find a feasible solution to

$$\text{(P7)} \quad \sum_{j \in T_i} \mathbf{x}_j - \sum_{j \in F_i} \mathbf{x}_j \geq 1 - |F_i| \quad i \in S$$
$$\mathbf{x}_j = 0 \quad \text{or} \quad 1 \quad \text{for } j = 1, 2, \ldots, n$$

By substituting $\mathbf{x}_j = 1 - \mathbf{y}_j$, where $\mathbf{y}_j = 0$ or 1, for $j \in F_i$, (P7) is equivalent to the set covering problem

$$\text{(P8)} \quad \text{Min} \sum_{j=1}^{n} (\mathbf{x}_j + \mathbf{y}_j) \tag{15.3}$$

$$\text{subject to} \quad \sum_{j \in T_i} \mathbf{x}_j + \sum_{j \in F_i} \mathbf{y}_j \geq 1 \qquad i \in S \tag{15.4}$$

$$\mathbf{x}_j + \mathbf{y}_j \geq 1 \quad j = 1, 2, \ldots, n \tag{15.5}$$

$$\mathbf{x}_j, \mathbf{y}_j = 0 \quad \text{or} \quad 1 \quad j = 1, 2, \ldots, n \tag{15.6}$$

Clearly (P7) is feasible if and only if (P8) has an optimal objective function value equal to n.

Given a set S of clauses and an additional clause $k \notin S$, the logical inference problem is to find out whether every truth assignment that satisfies all of the clauses in S also satisfies the clause k. The logical inference problem is

$$\text{(P9)} \quad \text{Min} \quad \sum_{j \in T_k} \mathbf{x}_j - \sum_{j \in F_k} \mathbf{x}_j$$

$$\text{subject to} \quad \sum_{j \in T_i} \mathbf{x}_j - \sum_{j \in F_i} \mathbf{x}_j \geq 1 - |F_i| \quad i \in S$$
$$\mathbf{x}_j = 0 \quad \text{or} \quad 1 \quad j = 1, 2, \ldots, n$$

The clause k is implied by the set of clauses S, if and only if (P9) has an optimal objective function value greater than $-|F_k|$. It is also straightforward to express the MAX-SAT problem (i.e., find a truth assignment that maximizes the number of satisfied clauses in a given set S) as an integer linear program.

15.4.1.5 Multiprocessor Scheduling

Given n jobs and m processors, the problem is to allocate each job to one and only one of the processors so as to minimize the make span time, i.e., minimize the completion time of all of the jobs. The processors may not be identical and, hence, job j if allocated to processor i requires p_{ij} units of time. The multiprocessor scheduling problem is

$$\text{(P10)} \quad \text{Min } T$$

$$\text{subject to} \quad \sum_{i=1}^{m} \mathbf{x}_{ij} = 1 \quad j = 1, 2, \ldots, n$$

$$\sum_{j=1}^{n} \mathbf{p}_{ij} \mathbf{x}_{ij} - T \leq 0 \quad i = 1, 2, \ldots, m$$

$$\mathbf{x}_{ij} = 0 \quad \text{or} \quad 1$$

Note that if all $\mathbf{p}_{ij}$ are integers, the optimal solution will be such that T is an integer.

15.4.2 Jeroslow's Representability Theorem

Jeroslow [1989], building on joint work with Lowe in 1984, characterized subsets of n-space that can be represented as the feasible region of a mixed integer (Boolean) program. They proved that a set is the feasible region of some mixed integer/linear programming problem (MILP) if and only if it is the union of finitely many polyhedra having the same recession cone (defined subsequently). Although this result is not widely known, it might well be regarded as the fundamental theorem of mixed integer modeling.

The basic idea of Jeroslow's results is that any set that can be represented in a mixed integer model can be represented in a disjunctive programming problem (i.e., a problem with either/or constraints). A *recession direction* for a set S in n-space is a vector $\mathbf{x}$ such that $s + \alpha\mathbf{x} \in S$ for all $s \in S$ and all $\alpha \geq 0$. The set of recession directions is denoted $rec(S)$. Consider the general mixed integer constraint set

$$\begin{gathered} \mathbf{f}(\mathbf{x}, \mathbf{y}, \boldsymbol{\lambda}) \leq \mathbf{b} \\ \mathbf{x} \in \Re^n, \qquad \mathbf{y} \in \Re^p \\ \boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_k), \quad \text{with} \quad \boldsymbol{\lambda}_j \in \{0, 1\} \quad \text{for } j = 1, \ldots, k \end{gathered} \tag{15.7}$$

Here $\mathbf{f}$ is a vector-valued function, so that $\mathbf{f}(\mathbf{x}, \mathbf{y}, \boldsymbol{\lambda}) \leq \mathbf{b}$ represents a set of constraints. We say that a set $S \subset \Re^n$ is *represented* by Eq. (15.6) if,

$$\mathbf{x} \in S \quad \text{if and only if } (\mathbf{x}, \mathbf{y}, \boldsymbol{\lambda}) \text{ satisfies Eq. (15.6) for some } y, \lambda.$$

If $\mathbf{f}$ is a linear transformation, so that Equation 15.6 is a MILP constraint set, we will say that S is *MILP representable*. The main result can now be stated.

Theorem 15.2 [Jeroslow and Lowe 1984, Jeroslow 1989]. *A set in n-space is MILP representable if and only if it is the union of finitely many polyhedra having the same set of recession directions.*

15.4.3 Benders's Representation

Any mixed integer linear program can be reformulated so that there is only one continuous variable. This reformulation, due to Benders [1962], will in general have an exponential number of constraints. Analogous to column generation, discussed earlier, these rows (constraints) can be generated as and when required.

Consider the (MILP)

$$\max\{\mathbf{cx} + \mathbf{dy} : A\mathbf{x} + G\mathbf{y} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}, \mathbf{y} \geq \mathbf{0} \text{ and integer}\}$$

Suppose the integer variables $\mathbf{y}$ are fixed at some values, then the associated linear program is

$$\text{(LP)} \quad \max\{\mathbf{cx} : \mathbf{x} \in \mathcal{P} = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b} - G\mathbf{y}, \mathbf{x} \geq \mathbf{0}\}\}$$

and its dual is

$$\text{(DLP)} \quad \min\{\mathbf{w}(\mathbf{b} - G\mathbf{y}) : \mathbf{w} \in \mathcal{Q} = \{\mathbf{w} : \mathbf{w}A \geq \mathbf{c}, \mathbf{w} \geq \mathbf{0}\}\}$$

Let $\{\mathbf{w}^k\}$, $k = 1, 2, \ldots, K$ be the extreme points of Q and $\{\mathbf{u}^j\}$, $j = 1, 2, \ldots, J$ be the extreme rays of the recession cone of Q, $\mathcal{C}_Q = \{\mathbf{u} : \mathbf{u}A \geq \mathbf{0}, \mathbf{u} \geq \mathbf{0}\}$. Note that if Q is nonempty, the $\{\mathbf{u}^j\}$ are all of the extreme rays of Q.

From linear programming duality, we know that if Q is empty and $\mathbf{u}^j(\mathbf{b} - G\mathbf{y}) \geq 0$, $j = 1, 2, \ldots, J$ for some $\mathbf{y} \geq \mathbf{0}$ and integer then (LP) and consequently (MILP) have an unbounded solution. If Q is nonempty and $\mathbf{u}^j(\mathbf{b} - G\mathbf{y}) \geq 0$, $j = 1, 2, \ldots, J$ for some $\mathbf{y} \geq \mathbf{0}$ and integer then (LP) has a finite optimum given by

$$\min_k \{\mathbf{w}^k(\mathbf{b} - G\mathbf{y})\}$$

Hence an equivalent formulation of (MILP) is

$$\begin{aligned}
\text{Max}\ \ & \alpha \\
& \alpha \le \mathbf{dy} + \mathbf{w}^k(\mathbf{b} - G\mathbf{y}), \quad k = 1, 2, \dots, K \\
\mathbf{u}^j(\mathbf{b} - G\mathbf{y}) &\ge 0, \quad j = 1, 2, \dots, J \\
& y \ge \mathbf{0} \text{ and integer} \\
& \alpha \quad \text{unrestricted}
\end{aligned}$$

which has only one continuous variable α as promised.

15.5 Polyhedral Combinatorics

One of the main purposes of writing down an algebraic formulation of a combinatorial optimization problem as an integer program is to then examine the linear programming relaxation and understand how well it represents the discrete integer program. There are somewhat special but rich classes of such formulations for which the linear programming relaxation is sharp or tight. These correspond to linear programs that have integer valued extreme points. Such polyhedra are called **integral polyhedra**.

15.5.1 Special Structures and Integral Polyhedra

A natural question of interest is whether the LP associated with an ILP has only integral extreme points. For instance, the linear programs associated with matching and edge covering polytopes in a bipartite graph have only integral vertices. Clearly, in such a situation, the ILP can be solved as LP. A polyhedron or a polytope is referred to as being integral if it is either empty or has only integral vertices.

Definition 15.3 A $0, \pm 1$ matrix is totally unimodular if the determinant of every square submatrix is 0 or ± 1.

Theorem 15.3 [Hoffman and Kruskal 1956]. *Let*

$$A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}$$

be a $0, \pm 1$ *matrix and*

$$\mathbf{b} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{pmatrix}$$

be a vector of appropriate dimensions. Then A is totally unimodular if and only if the polyhedron

$$P(A, \mathbf{b}) = \{\mathbf{x} : A_1\mathbf{x} \le \mathbf{b}_1; A_2\mathbf{x} \ge \mathbf{b}_2; A_3\mathbf{x} = \mathbf{b}_3; \mathbf{x} \ge 0\}$$

is integral for all integral vectors $\mathbf{b}$.

The constraint matrix associated with a network flow problem (see, for instance, Ahuja et al. [1993]) is totally unimodular. Note that for a given integral $\mathbf{b}$, $P(A, \mathbf{b})$ may be integral even if A is not totally unimodular.

Definition 15.4 A polyhedron defined by a system of linear constraints is totally dual integral (TDI) if for each objective function with integral coefficient the dual linear program has an integral optimal solution whenever an optimal solution exists.

Theorem 15.4 [Edmonds and Giles 1977]. *If $P(A) = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}\}$ is TDI and $\mathbf{b}$ is integral, then $P(A)$ is integral.*

Hoffman and Kruskal [1956] have, in fact, shown that the polyhedron $P(A, \mathbf{b})$ defined in Theorem 15.3 is TDI. This follows from Theorem 15.3 and the fact that A is totally unimodular if and only if A^T is totally unimodular.

Balanced matrices, first introduced by Berge [1972] have important implications for packing and covering problems (see also Berge and Las Vergnas [1970]).

Definition 15.5 A 0, 1 matrix is balanced if it does not contain a square submatrix of odd order with two ones per row and column.

Theorem 15.5 [Berge 1972, Fulkerson et al. 1974]. *Let A be a balanced 0, 1 matrix. Then the set packing, set covering, and set partitioning polytopes associated with A are integral, i.e., the polytopes*

$$P(A) = \{\mathbf{x} : \mathbf{x} \geq \mathbf{0};\ A\mathbf{x} \leq \mathbf{1}\}$$
$$Q(A) = \{\mathbf{x} : \mathbf{0} \leq \mathbf{x} \leq \mathbf{1};\ A\mathbf{x} \geq \mathbf{1}\}$$
$$R(A) = \{\mathbf{x} : \mathbf{x} \geq \mathbf{0};\ A\mathbf{x} = \mathbf{1}\}$$

are integral.

Let

$$A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}$$

be a balanced 0, 1 matrix. Fulkerson et al. [1974] have shown that the polytope $P(A) = \{\mathbf{x} : A_1\mathbf{x} \leq \mathbf{1}; A_2\mathbf{x} \geq \mathbf{1}; A_3\mathbf{x} = \mathbf{1}; \mathbf{x} \geq \mathbf{0}\}$ is TDI and by the theorem of Edmonds and Giles [1977] it follows that $P(A)$ is integral.

Truemper [1992] has extended the definition of balanced matrices to include $0, \pm 1$ matrices.

Definition 15.6 A $0, \pm 1$ matrix is balanced if for every square submatrix with exactly two nonzero entries in each row and each column, the sum of the entries is a multiple of 4.

Theorem 15.6 [Conforti and Cornuejols 1992b]. *Suppose A is a balanced $0, \pm 1$ matrix. Let $\mathbf{n}(A)$ denote the column vector whose ith component is the number of -1s in the ith row of A. Then the polytopes*

$$P(A) = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$
$$Q(A) = \{\mathbf{x} : A\mathbf{x} \geq \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$
$$R(A) = \{\mathbf{x} : A\mathbf{x} = \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$

are integral.

Note that a $0, \pm 1$ matrix A is balanced if and only if A^T is balanced. Moreover, A is balanced (totally unimodular) if and only if every submatrix of A is balanced (totally unimodular). Thus, if A is balanced (totally unimodular) it follows that Theorem 15.6 (Theorem 15.3) holds for every submatrix of A.

Totally unimodular matrices constitute a subclass of balanced matrices, i.e., a totally unimodular $0, \pm 1$ matrix is always balanced. This follows from a theorem of Camion [1965], which states that a $0, \pm 1$ is totally unimodular if and only if for every square submatrix with an even number of nonzero entries in each row and in each column, the sum of the entries equals a multiple of 4. The 4×4 matrix in Figure 15.1

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

FIGURE 15.1 A balanced matrix and a perfect matrix. (From Chandru, V. and Rao, M. R. Combinatorial optimization: an integer programming perspective. ACM *Comput. Surveys*, 28, 1. March 1996.)

illustrates the fact that a balanced matrix is not necessarily totally unimodular. Balanced $0, \pm 1$ matrices have implications for solving the satisfiability problem. If the given set of clauses defines a balanced $0, \pm 1$ matrix, then as shown by Conforti and Cornuejols [1992b], the satisfiability problem is trivial to solve and the associated MAXSAT problem is solvable in polynomial time by linear programming. A survey of balanced matrices is in Conforti et al. [1994].

Definition 15.7 A $0, 1$ matrix A is perfect if the set packing polytope $P(A) = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{1}; \mathbf{x} \geq \mathbf{0}\}$ is integral.

The chromatic number of a graph is the minimum number of colors required to color the vertices of the graph so that no two vertices with the same color have an edge incident between them. A graph G is perfect if for every node induced subgraph H, the chromatic number of H equals the number of nodes in the maximum clique of H. The connections between the integrality of the set packing polytope and the notion of a perfect graph, as defined by Berge [1961, 1970], are given in Fulkerson [1970], Lovasz [1972], Padberg [1974], and Chvátal [1975].

Theorem 15.7 [Fulkerson 1970, Lovasz 1972, Chvátal 1975] *Let A be $0, 1$ matrix whose columns correspond to the nodes of a graph G and whose rows are the incidence vectors of the maximal cliques of G. The graph G is perfect if and only if A is perfect.*

Let G_A denote the intersection graph associated with a given $0, 1$ matrix A (see Section 15.4). Clearly, a row of A is the incidence vector of a clique in G_A. In order for A to be perfect, every maximal clique of G_A must be represented as a row of A because inequalities defined by maximal cliques are facet defining. Thus, by Theorem 15.7, it follows that a $0, 1$ matrix A is perfect if and only if the undominated (a row of A is dominated if its support is contained in the support of another row of A) rows of A form the clique-node incidence matrix of a perfect graph.

Balanced matrices with $0, 1$ entries, constitute a subclass of $0, 1$ perfect matrices, i.e., if a $0, 1$ matrix A is balanced, then A is perfect. The 4×3 matrix in Figure 15.1 is an example of a matrix that is perfect but not balanced.

Definition 15.8 A $0, 1$ matrix A is ideal if the set covering polytope

$$Q(A) = \{\mathbf{x} \;:\; A\mathbf{x} \geq \mathbf{1}; \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$

is integral.

Properties of ideal matrices are described by Lehman [1979], Padberg [1993], and Cornuejols and Novick [1994]. The notion of a $0, 1$ perfect (ideal) matrix has a natural extension to a $0, \pm 1$ perfect (ideal) matrix. Some results pertaining to $0, \pm 1$ ideal matrices are contained in Hooker [1992], whereas some results pertaining to $0, \pm 1$ perfect matrices are given in Conforti et al. [1993].

An interesting combinatorial problem is to check whether a given $0, \pm 1$ matrix is totally unimodular, balanced, or perfect. Seymour's [1980] characterization of totally unimodular matrices provides a polynomial-time algorithm to test whether a given matrix $0, 1$ matrix is totally unimodular. Conforti

et al. [1999] give a polynomial-time algorithm to check whether a $0, 1$ matrix is balanced. This has been extended by Conforti et al. [1994] to check in polynomial time whether a $0, \pm 1$ matrix is balanced. An open problem is that of checking in polynomial time whether a $0, 1$ matrix is perfect. For linear matrices (a matrix is linear if it does not contain a 2×2 submatrix of all ones), this problem has been solved by Fonlupt and Zemirline [1981] and Conforti and Rao [1993].

15.5.2 Matroids

Matroids and submodular functions have been studied extensively, especially from the point of view of combinatorial optimization (see, for instance, Nemhauser and Wolsey [1988]). Matroids have nice properties that lead to efficient algorithms for the associated optimization problems. One of the interesting examples of a matroid is the problem of finding a maximum or minimum weight spanning tree in a graph. Two different but equivalent definitions of a matroid are given first. A greedy algorithm to solve a linear optimization problem over a matroid is presented. The matroid intersection problem is then discussed briefly.

Definition 15.9 Let $N = \{1, 2, \cdot, n\}$ be a finite set and let $\mathcal{F}$ be a set of subsets of N. Then $I = (N, \mathcal{F})$ is an independence system if $S_1 \in \mathcal{F}$ implies that $S_2 \in \mathcal{F}$ for all $S_2 \subseteq S_1$. Elements of $\mathcal{F}$ are called independent sets. A set $S \in \mathcal{F}$ is a maximal independent set if $S \cup \{j\} \notin \mathcal{F}$ for all $j \in N \backslash S$. A maximal independent set T is a maximum if $|T| \geq |S|$ for all $S \in \mathcal{F}$.

The rank $r(Y)$ of a subset $Y \subseteq N$ is the cardinality of the maximum independent subset $X \subseteq Y$. Note that $r(\phi) = 0$, $r(X) \leq |X|$ for $X \subseteq N$ and the rank function is nondecreasing, i.e., $r(X) \leq r(Y)$ for $X \subseteq Y \subseteq N$.

A matroid $M = (N, \mathcal{F})$ is an independence system in which every maximal independent set is a maximum.

Example 15.2

Let $G = (V, E)$ be an undirected connected graph with V as the node set and E as the edge set.

1. Let $I = (E, \mathcal{F})$ where $F \in \mathcal{F}$ if $F \subseteq E$ is such that at most one edge in F is incident to each node of V, that is, $F \in \mathcal{F}$ if F is a matching in G. Then $I = (E, \mathcal{F})$ is an independence system but not a matroid.
2. Let $M = (E, \mathcal{F})$ where $F \in \mathcal{F}$ if $F \subseteq E$ is such that $G_F = (V, F)$ is a forest, that is, G_F contains no cycles. Then $M = (E, \mathcal{F})$ is a matroid and maximal independent sets of M are spanning trees.

An alternative but equivalent definition of matroids is in terms of submodular functions.

Definition 15.10 A nondecreasing integer valued submodular function r defined on the subsets of N is called a matroid rank function if $r(\phi) = 0$ and $r(\{j\}) \leq 1$ for $j \in N$. The pair (N, r) is called a matroid.

A nondecreasing, integer-valued, submodular function f, defined on the subsets of N is called a polymatroid function if $f(\phi) = 0$. The pair (N, r) is called a polymatroid.

15.5.2.1 Matroid Optimization

To decide whether an optimization problem over a matroid is polynomially solvable or not, we need to first address the issue of representation of a matroid. If the matroid is given either by listing the independent sets or by its rank function, many of the associated linear optimization problems are trivial to solve. However, matroids associated with graphs are completely described by the graph and the condition for independence. For instance, the matroid in which the maximal independent sets are spanning forests, the graph $G = (V, E)$ and the independence condition of no cycles describes the matroid.

Most of the algorithms for matroid optimization problems require a test to determine whether a specified subset is independent. We assume the existence of an oracle or subroutine to do this checking in running time, which is a polynomial function of $|N| = n$.

Maximum Weight Independent Set. Given a matroid $M = (N, \mathcal{F})$ and weights w_j for $j \in N$, the problem of finding a maximum weight independent set is $\max_{F \in \mathcal{F}} \left\{ \sum_{j \in F} w_j \right\}$. The greedy algorithm to solve this problem is as follows:

Procedure 15.3 Greedy:

0. **Initialize:** Order the elements of N so that $w_i \geq w_{i+1}$, $i = 1, 2, \ldots, n-1$. Let $T = \phi, i = 1$.
1. **If** $w_i \leq 0$ or $i > n$, **stop** T is optimal, i.e., $x_j = 1$ for $j \in T$ and $x_j = 0$ for $j \notin T$. If $w_i > 0$ and $T \cup \{i\} \in \mathcal{F}$, add element i to T.
2. **Increment** i by 1 and return to step 1.

Edmonds [1970, 1971] derived a complete description of the *matroid polytope*, the convex hull of the characteristic vectors of independent sets of a matroid. While this description has a large (exponential) number of constraints, it permits the treatment of linear optimization problems on independent sets of matroids as linear programs. Cunningham [1984] describes a polynomial algorithm to solve the separation problem for the matroid polytope. The matroid polytope and the associated greedy algorithm have been extended to polymatroids (Edmonds [1970], McDiarmid [1975]).

The separation problem for a polymatroid is equivalent to the problem of minimizing a submodular function defined over the subsets of N (see Nemhauser and Wolsey [1988]). A class of submodular functions that have some additional properties can be minimized in polynomial time by solving a maximum flow problem [Rhys 1970, Picard and Ratliff 1975]. The general submodular function can be minimized in polynomial time by the ellipsoid algorithm [Grötschel et al. 1988].

The uncapacitated plant location problem formulated in Section 15.4 can be reduced to maximizing a submodular function. Hence, it follows that maximizing a submodular function is $\mathcal{NP}$-hard.

15.5.2.2 Matroid Intersection

A matroid intersection problem involves finding an independent set contained in two or more matroids defined on the same set of elements.

Let $G = (V_1, V_2, E)$ be a bipartite graph. Let $M_i = (E, \mathcal{F}_i), i = 1, 2$, where $F \in \mathcal{F}_i$ if $F \subseteq E$ is such that no more than one edge of F is incident to each node in V_i. The set of matchings in G constitutes the intersection of the two matroids $M_i, i = 1, 2$. The problem of finding a maximum weight independent set in the intersection of two matroids can be solved in polynomial time [Lawler 1975, Edmonds 1970, 1979, Frank 1981]. The two (poly) matroid intersection polytope has been studied by Edmonds [1979].

The problem of testing whether a graph contains a Hamiltonian path is $\mathcal{NP}$-complete. Since this problem can be reduced to the problem of finding a maximum cordinality independent set in the intersection of three matroids, it follows that the matroid intersection problem involving three or more matroids is $\mathcal{NP}$-hard.

15.5.3 Valid Inequalities, Facets, and Cutting Plane Methods

Earlier in this section, we were concerned with conditions under which the packing and covering polytopes are integral. But, in general, these polytopes are not integral, and additional inequalities are required to have a complete linear description of the convex hull of integer solutions. The existence of finitely many such linear inequalities is guaranteed by Weyl's [1935] Theorem.

Consider the feasible region of an ILP given by

$$P_I = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}; \mathbf{x} \geq \mathbf{0} \text{ and integer}\} \tag{15.8}$$

Recall that an inequality $\mathbf{fx} \leq f_0$ is referred to as a valid inequality for P_I if $\mathbf{fx}^* \leq f_0$ for all $\mathbf{x}^* \in P_I$. A valid linear inequality for $P_I(A, \mathbf{b})$ is said to be facet defining if it intersects $P_I(A, \mathbf{b})$ in a face of dimension one less than the dimension of $P_I(A, \mathbf{b})$. In the example shown in Figure 15.2, the inequality $\mathbf{x}_2 + \mathbf{x}_3 \leq 1$ is a facet defining inequality of the integer hull.

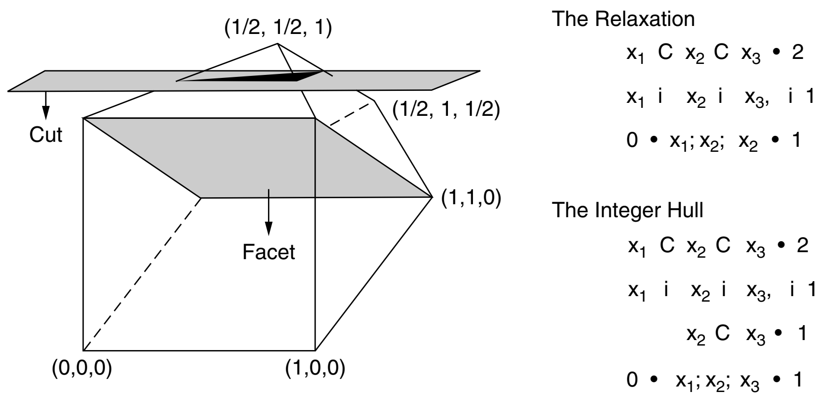


FIGURE 15.2 Relaxation, cuts, and facets (From Chandru, V. and Rao, M. R. Combinatorial optimization: an integer programming perspective. *ACM Comput. Surveys,* 28, 1. March 1996.)

Let $\mathbf{u} \geq 0$ be a row vector of appropriate size. Clearly $\mathbf{u}A\mathbf{x} \leq \mathbf{ub}$ holds for every $\mathbf{x}$ in P_I. Let $(\mathbf{u}A)_j$ denote the jth component of the row vector $\mathbf{u}A$ and $\lfloor(\mathbf{u}A)_j\rfloor$ denote the largest integer less than or equal to $(\mathbf{u}A)_j$. Now, since $\mathbf{x} \in P_I$ is a vector of nonnegative integers, it follows that $\sum_j \lfloor(\mathbf{u}A)_j\rfloor\mathbf{x}_j \leq \lfloor\mathbf{ub}\rfloor$ is a valid inequality for P_I. This scheme can be used to generate many valid inequalities by using different $\mathbf{u} \geq \mathbf{0}$. Any set of generated valid inequalities may be added to the constraints in Equation 15.7 and the process of generating them may be repeated with the enhanced set of inequalities. This iterative procedure of generating valid inequalities is called Gomory–Chvátal (GC) rounding. It is remarkable that this simple scheme is complete, i.e., every valid inequality of P_I can be generated by finite application of GC rounding (Chvátal [1973], Schrijver [1986]).

The number of inequalities needed to describe the convex hull of P_I is usually exponential in the size of A. But to solve an optimization problem on P_I, one is only interested in obtaining a partial description of P_I that facilitates the identification of an integer solution and prove its optimality. This is the underlying basis of any cutting plane approach to combinatorial problems.

15.5.3.1 The Cutting Plane Method

Consider the optimization problem

$$\max\{\mathbf{cx} : \mathbf{x} \in P_I = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}; \mathbf{x} \geq \mathbf{0} \text{ and integer}\}\}$$

The generic **cutting plane** method as applied to this formulation is given as follows.

Procedure 15.4 Cutting Plane:

1. Initialize $A' \leftarrow A$ and $\mathbf{b}' \leftarrow \mathbf{b}$.
2. Find an optimal solution $\bar{\mathbf{x}}$ to the linear program

$$\max\{\mathbf{cx} : A'\mathbf{x} \leq \mathbf{b}'; \mathbf{x} \geq \mathbf{0}\}$$

   If $\bar{\mathbf{x}} \in P_I$, stop and return $\bar{\mathbf{x}}$.
3. Generate a valid inequality $\mathbf{fx} \leq f_0$ for P_I such that $\mathbf{f}\bar{\mathbf{x}} > f_0$ (the inequality "cuts" $\bar{\mathbf{x}}$).
4. Add the inequality to the constraint system, update

$$A' \leftarrow \begin{pmatrix} A' \\ \mathbf{f} \end{pmatrix}, \qquad \mathbf{b}' \leftarrow \begin{pmatrix} \mathbf{b}' \\ f_0 \end{pmatrix}$$

   Go to step 2.

In step 3 of the cutting plane method, we require a suitable application of the GC rounding scheme (or some alternative method of identifying a cutting plane). Notice that while the GC rounding scheme will generate valid inequalities, the identification of one that cuts off the current solution to the linear programming relaxation is all that is needed. Gomory [1958] provided just such a specialization of the rounding scheme that generates a cutting plane. Although this met the theoretical challenge of designing a sound and complete cutting plane method for integer linear programming, it turned out to be a weak method in practice. Successful cutting plane methods, in use today, use considerable additional insights into the structure of facet-defining cutting planes. Using facet cuts makes a huge difference in the speed of convergence of these methods. Also, the idea of combining cutting plane methods with search methods has been found to have a lot of merit. These branch and cut methods will be discussed in the next section.

15.5.3.2 The *b*-Matching Problem

Consider the **b**-matching problem:

$$\max\{\mathbf{cx} : A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0} \text{ and integer}\} \tag{15.9}$$

where A is the node-edge incidence matrix of an undirected graph and **b** is a vector of positive integers. Let G be the undirected graph whose node-edge incidence matrix is given by A and let $W \subseteq V$ be any subset of nodes of G (i.e., subset of rows of A) such that

$$\mathbf{b}(W) = \sum_{i \in W} \mathbf{b}_i$$

is odd. Then the inequality

$$\mathbf{x}(W) = \sum_{e \in E(W)} \mathbf{x}_e \leq \frac{1}{2}(\mathbf{b}(W) - 1) \tag{15.10}$$

is a valid inequality for integer solutions to Equation 15.8 where $E(W) \subseteq E$ is the set of edges of G having both ends in W. Edmonds [1965] has shown that the inequalities Equation 15.8 and Equation 15.9 define the integral **b**-matching polytope. Note that the number of inequalities Equation 15.9 is exponential in the number of nodes of G. An instance of the successful application of the idea of using only a partial description of P_I is in the blossom algorithm for the matching problem, due to Edmonds [1965].

As we saw, an implication of the ellipsoid method for linear programming is that the linear program over P_I can be solved in polynomial time if and only if the associated separation problem (also referred to as the constraint identification problem, see Section 15.2) can be solved in polynomial time, see Grötschel et al. [1982], Karp and Papadimitriou [1982], and Padberg and Rao [1981]. The separation problem for the **b**-matching problem with or without upper bounds was shown by Padberg and Rao [1982], to be solvable in polynomial time. The procedure involves a minor modification of the algorithm of Gomory and Hu [1961] for multiterminal networks. However, no polynomial (in the number of nodes of the graph) linear programming formulation of this separation problem is known. A related unresolved issue is whether there exists a polynomial size (compact) formulation for the **b**-matching problem. Yannakakis [1988] has shown that, under a symmetry assumption, such a formulation is impossible.

15.5.3.3 Other Combinatorial Problems

Besides the matching problem, several other combinatorial problems and their associated polytopes have been well studied and some families of facet defining inequalities have been identified. For instance, the set packing, graph partitioning, plant location, max cut, traveling salesman, and Steiner tree problems have been extensively studied from a polyhedral point of view (see, for instance, Nemhauser and Wolsey [1988]).

These combinatorial problems belong to the class of $\mathcal{NP}$-complete problems. In terms of a worst-case analysis, no polynomial-time algorithms are known for these problems. Nevertheless, using a cutting plane approach with branch and bound or branch and cut (see Section 15.6), large instances of these problems

have been successfully solved, see Crowder et al. [1983], for general $0-1$ problems, Barahona et al. [1989] for the max cut problem, Padberg and Rinaldi [1991] for the traveling salesman problem, and Chopra et al. [1992] for the Steiner tree problem.

15.6 Partial Enumeration Methods

In many instances, to find an optimal solution to integer linear programing problems (ILP), the structure of the problem is exploited together with some sort of partial enumeration. In this section, we review the branch and bound (B-and-B) and branch and cut (B-and-C) methods for solving an ILP.

15.6.1 Branch and Bound

The branch bound (B-and-B) method is a systematic scheme for implicitly enumerating the finitely many feasible solutions to an ILP. Although, theoretically the size of the enumeration tree is exponential in the problem parameters, in most cases, the method eliminates a large number of feasible solutions. The key features of branch and bound method are:

1. **Selection/removal** of one or more problems from a candidate list of problems
2. **Relaxation** of the selected problem so as to obtain a lower bound (on a minimization problem) on the optimal objective function value for the selected problem
3. **Fathoming**, if possible, of the selected problem
4. **Branching** strategy is needed if the selected problem is not fathomed. Branching creates subproblems, which are added to the candidate list of problems.

The four steps are repeated until the candidate list is empty. The B-and-B method sequentially examines problems that are added and removed from a candidate list of problems.

15.6.1.1 Initialization

Initially, the candidate list contains only the original ILP, which is denoted as

$$(P) \quad \min\{\mathbf{cx} : A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0} \text{ and integer}\}$$

Let $F(P)$ denote the feasible region of (P) and $z(P)$ denote the optimal objective function value of (P). For any $\bar{\mathbf{x}}$ in $F(P)$, let $z_P(\bar{\mathbf{x}}) = \mathbf{c}\bar{\mathbf{x}}$.

Frequently, heuristic procedures are first applied to get a good feasible solution to (P). The best solution known for (P) is referred to as the current incumbent solution. The corresponding objective function value is denoted as z_I. In most instances, the initial heuristic solution is neither optimal nor at least immediately certified to be optimal. Thus, further analysis is required to ensure that an optimal solution to (P) is obtained. If no feasible solution to (P) is known, z_I is set to ∞.

15.6.1.2 Selection/Removal

In each iterative step of B-and-B, a problem is selected and removed from the candidate list for further analysis. The selected problem is henceforth referred to as the candidate problem (*CP*). The algorithm terminates if there is no problem to select from the candidate list. Initially, there is no issue of selection since the candidate list contains only the problem (P). However, as the algorithm proceeds, there would be many problems on the candidate list and a selection rule is required. Appropriate selection rules, also referred to as branching strategies, are discussed later. Conceptually, several problems may be simultaneously selected and removed from the candidate list. However, most sequential implementations of B-and-B select only one problem from the candidate list and this is assumed henceforth. Parallel aspects of B-and-B on $0-1$ integer linear programs are discussed in Cannon and Hoffman [1990] and for the case of traveling salesman problems in Applegate et al. [1994].

The computational time required for the B-and-B algorithm depends crucially on the order in which the problems in the candidate list are examined. A number of clever heuristic rules may be employed in devising such strategies. Two general purpose selection strategies that are commonly used are as follows:

1. Choose the problem that was added last to the candidate list. This last-in–first-out rule (LIFO) is also called depth first search (DFS) since the selected candidate problem increases the depth of the active enumeration tree.
2. Choose the problem on the candidate list that has the least lower bound. Ties may be broken by choosing the problem that was added last to the candidate list. This rule would require that a lower bound be obtained for each of the problems on the candidate list. In other words, when a problem is added to the candidate list, an associated lower bound should also be stored. This may be accomplished by using ad hoc rules or by solving a relaxation of each problem before it is added to the candidate list.

Rule 1 is known to empirically dominate rule 2 when storage requirements for candidate list and computation time to solve (P) are taken into account. However, some analysis indicates that rule 2 can be shown to be superior if minimizing the number of candidate problems to be solved is the criterion (see Parker and Rardin [1988]).

15.6.1.3 Relaxation

In order to analyze the selected candidate problem (CP), a **relaxation** (CP_R) of (CP) is solved to obtain a lower bound $z(CP_R) \leq z(CP)$. (CP_R) is a relaxation of (CP) if:

1. $F(CP) \subseteq F(CP_R)$
2. For $\bar{\mathbf{x}} \in F(CP)$, $z_{CP_R}(\bar{\mathbf{x}}) \leq z_{CP}(\bar{\mathbf{x}})$
3. For $\bar{\mathbf{x}}, \hat{\mathbf{x}} \in F(CP)$, $z_{CP_R}(\bar{\mathbf{x}}) \leq z_{CP_R}(\hat{\mathbf{x}})$ implies that $z_{CP}(\bar{\mathbf{x}}) \leq z_{CP}(\hat{\mathbf{x}})$

Relaxations are needed because the candidate problems are typically hard to solve. The relaxations used most often are either linear programming or Lagrangian relaxations of (CP), see Section 15.7 for details. Sometimes, instead of solving a relaxation of (CP), a lower bound is obtained by using some ad hoc rules such as penalty functions.

15.6.1.4 Fathoming

A candidate problem is fathomed if:

(FC1) analysis of (CP_R) reveals that (CP) is infeasible. For instance, if $F(CP_R) = \phi$, then $F(CP) = \phi$.
(FC2) analysis of (CP_R) reveals that (CP) has no feasible solution better than the current incumbent solution. For instance, if $z(CP_R) \geq z_I$, then $z(CP) \geq z(CP_R) \geq z_I$.
(FC3) analysis of (CP_R) reveals an optimal solution of (CP). For instance, if the optimal solution, $\mathbf{x}_R$, to (CP_R) is feasible in (CP), then $(\mathbf{x}_R)$ is an optimal solution to (CP) and $z(CP) = \mathbf{c}\mathbf{x}_R$.
(FC4) analysis of (CP_R) reveals that (CP) is dominated by some other problem, say, CP^*, in the candidate list. For instance, if it can shown that $z(CP^*) \leq z(CP)$, then there is no need to analyze (CP) further.

If a candidate problem (CP) is fathomed using any of the preceding criteria, then further examination of (CP) or its descendants (subproblems) obtained by separation is not required. If (FC3) holds, and $z(CP) < z_I$, the incumbent is updated as $\mathbf{x}_R$ and z_I is updated as $z(CP)$.

15.6.1.5 Separation/Branching

If the candidate problem (CP) is not fathomed, then CP is separated into several problems, say, (CP_1), $(CP_2), \ldots, (CP_q)$, where $\bigcup_{t=1}^{q} F(CP_t) = F(CP)$ and, typically,

$$F(CP_i) \cap F(CP_j) = \phi \; \forall \, i \neq j$$

For instance, a separation of (CP) into $(CP_i), i = 1, 2, \ldots, q$, is obtained by fixing a single variable, say, $\mathbf{x}_j$, to one of the q possible values of $\mathbf{x}_j$ in an optimal solution to (CP). The choice of the variable

to fix depends on the separation strategy, which is also part of the branching strategy. After separation, the subproblems are added to the candidate list. Each subproblem (CP_t) is a restriction of (CP) since $F(CP_t) \subseteq F(CP)$. Consequently, $z(CP) \leq z(CP_t)$ and $z(CP) = \min_t z(CP_t)$.

The various steps in the B-and-B algorithm are outlined as follows.

Procedure 15.5 B-and-B:

0. **Initialize:** Given the problem (P), the incumbent value z_I is obtained by applying some heuristic (if a feasible solution to (P) is not available, set $z_I = +\infty$). Initialize the candidate list $C \leftarrow \{(P)\}$.
1. **Optimality:** If $C = \emptyset$ and $z_I = +\infty$, then (P) is infeasible, stop. Stop also if $C = \emptyset$ and $z_I < +\infty$, the incumbent is an optimal solution to (P).
2. **Selection:** Using some candidate selection rule, select and remove a problem $(CP) \in C$.
3. **Bound:** Obtain a lower bound for (CP) by either solving a relaxation (CP_R) of (CP) or by applying some ad-hoc rules. If (CP_R) is infeasible, return to Step 1. Else, let x_R be an optimal solution of (CP_R).
4. **Fathom:** If $z(CP_R) \geq z_I$, return to step 1. Else if x_R is feasible in (CP) and $z(CP) < z_I$, set $z_I \leftarrow z(CP)$, update the incumbent as x_R and return to step 1. Finally, if x_R is feasible in (CP) but $z(CP) \geq z_I$, return to step 1.
5. **Separation:** Using some separation or branching rule, separate (CP) into $(CP_i), i = 1, 2, \ldots, q$ and set $C \leftarrow C \cup \{CP_1), (CP_2), \ldots, (CP_q)\}$ and return to step 1.
6. **End Procedure.**

Although the B-and-B method is easy to understand, the implementation of this scheme for a particular ILP is a nontrivial task requiring the following:

1. A relaxation strategy with efficient procedures for solving these relaxations
2. Efficient data-structures for handling the rather complicated bookkeeping of the candidate list
3. Clever strategies for selecting promising candidate problems
4. Separation or branching strategies that could effectively prune the enumeration tree

A key problem is that of devising a relaxation strategy, that is, to find *good relaxations*, which are significantly easier to solve than the original problems and tend to give sharp lower bounds. Since these two are conflicting, one has to find a reasonable tradeoff.

15.6.2 Branch and Cut

In the past few years, the branch and cut (B-and-C) method has become popular for solving NP-complete combinatorial optimization problems. As the name suggests, the B-and-C method incorporates the features of both the branch and bound method just presented and the cutting plane method presented previously. The main difference between the B-and-C method and the general B-and-B scheme is in the bound step (step 3).

A distinguishing feature of the B-and-C method is that the relaxation (CP_R) of the candidate problem (CP) is a linear programming problem, and, instead of merely solving (CP_R), an attempt is made to solve (CP) by using cutting planes to tighten the relaxation. If (CP_R) contains inequalities that are valid for (CP) but not for the given ILP, then the GC rounding procedure may generate inequalities that are valid for (CP) but not for the ILP. In the B-and-C method, the inequalities that are generated are always valid for the ILP and hence can be used globally in the enumeration tree.

Another feature of the B-and-C method is that often heuristic methods are used to convert some of the fractional solutions, encountered during the cutting plane phase, into feasible solutions of the (CP) or more generally of the given ILP. Such feasible solutions naturally provide upper bounds for the ILP. Some of these upper bounds may be better than the previously identified best upper bound and, if so, the current incumbent is updated accordingly.

We thus obtain the B-and-C method by replacing the bound step (step 3) of the B-and-B method by steps 3(a) and 3(b) and also by replacing the fathom step (step 4) by steps 4(a) and 4(b) given subsequently.

3(a) **Bound:** Let (CP_R) be the LP relaxation of (CP). Attempt to solve (CP) by a cutting plane method which generates valid inequalities for (P). Update the constraint system of (P) and the incumbent as appropriate.

Let $F\mathbf{x} \leq \mathbf{f}$ denote all of the valid inequalities generated during this phase. Update the constraint system of (P) to include all of the generated inequalities, i.e., set $A^T \leftarrow (A^T, F^T)$ and $\mathbf{b}^T \leftarrow (\mathbf{b}^T, \mathbf{f}^T)$. The constraints for all of the problems in the candidate list are also to be updated.

During the cutting plane phase, apply heuristic methods to convert some of the identified fractional solutions into feasible solutions to (P). If a feasible solution, $\bar{\mathbf{x}}$, to (P), is obtained such that $\mathbf{c}\bar{\mathbf{x}} < z_I$, update the incumbent to $\bar{\mathbf{x}}$ and z_I to $\mathbf{c}\bar{\mathbf{x}}$. Hence, the remaining changes to B-and-B are as follows:

3(b) **If** (CP) is solved go to step 4(a). **Else**, let $\hat{\mathbf{x}}$ be the solution obtained when the cutting plane phase is terminated, (we are unable to identify a valid inequality of (P) that is violated by $\hat{\mathbf{x}}$). Go to step 4(b).

4(a) **Fathom by Optimality:** Let $\mathbf{x}^*$ be an optimal solution to (CP). If $z(CP) < z_I$, set $\mathbf{x}_I \leftarrow z(CP)$ and update the incumbent as $\mathbf{x}^*$. Return to step 1.

4(b) **Fathom by Bound:** If $\mathbf{c}\hat{\mathbf{x}} \geq z_I$, return to Step 1.
Else go to step 5.

The incorporation of a cutting plane phase into the B-and-B scheme involves several technicalities which require careful design and implementation of the B-and-C algorithm. Details of the state of the art in cutting plane algorithms including the B-and-C algorithm are reviewed in Jünger et al. [1995].

15.7 Approximation in Combinatorial Optimization

The inherent complexity of integer linear programming has led to a long-standing research program in approximation methods for these problems. Linear programming relaxation and Lagrangian relaxation are two general approximation schemes that have been the real workhorses of computational practice. Semidefinite relaxation is a recent entrant that appears to be very promising. In this section, we present a brief review of these developments in the approximation of combinatorial optimization problems.

In the past few years, there has been significant progress in our understanding of performance guarantees for approximation of $\mathcal{NP}$-hard combinatorial optimization problems. A **ρ-approximate** algorithm for an optimization problem is an approximation algorithm that delivers a feasible solution with objective value within a factor of ρ of optimal (think of minimization problems and $\rho \geq 1$). For some combinatorial optimization problems, it is possible to *efficiently* find solutions that are arbitrarily close to optimal even though finding the true optimal is hard. If this were true of most of the problems of interest, we would be in good shape. However, the recent results of Arora et al. [1992] indicate exactly the opposite conclusion.

A polynomial-time approximation scheme (PTAS) for an optimization problem is a family of algorithms, A_ρ, such that for each $\rho > 1$, A_ρ is a polynomial-time ρ-approximate algorithm. Despite concentrated effort spanning about two decades, the situation in the early 1990s was that for many combinatorial optimization problems, we had no PTAS and no evidence to suggest the nonexistence of such schemes either. This led Papadimitriou and Yannakakis [1991] to define a new complexity class (using reductions that preserve approximate solutions) called MAXSNP, and they identified several complete languages in this class. The work of Arora et al. [1992] completed this agenda by showing that, assuming $\mathcal{P} \neq \mathcal{NP}$, there is no PTAS for a MAXSNP-complete problem.

An implication of these theoretical developments is that for most combinatorial optimization problems, we have to be quite satisfied with performance guarantee factors ρ that are of some small fixed value. (There are problems, like the general traveling salesman problem, for which there are no ρ-approximate algorithms

for any finite value of ρ, assuming of course that $\mathcal{P} \neq \mathcal{NP}$.) Thus, one avenue of research is to go problem by problem and knock ρ down to its smallest possible value. A different approach would be to look for other notions of good approximations based on probabilistic guarantees or empirical validation. Let us see how the polyhedral combinatorics perspective helps in each of these directions.

15.7.1 LP Relaxation and Randomized Rounding

Consider the well-known problem of finding the *smallest weight vertex cover* in a graph. We are given a graph $G(V, E)$ and a nonnegative weight $\mathbf{w}(v)$ for each vertex $v \in V$. We want to find the smallest total weight subset of vertices S such that each edge of G has at least one end in S. (This problem is known to be MAXSNP-hard.) An integer programming formulation of this problem is given by

$$\min\left\{\sum_{v\in V}\mathbf{w}(v)\mathbf{x}(v) : \mathbf{x}(u)+\mathbf{x}(v)\geq 1,\ \forall(u,v)\in E,\ \ \mathbf{x}(v)\in\{0,1\}\ \forall v\in V\right\}$$

To obtain the linear programming relaxation we substitute the $\mathbf{x}(v) \in \{0, 1\}$ constraint with $\mathbf{x}(v) \geq 0$ for each $v \in V$. Let $\mathbf{x}^*$ denote an optimal solution to this relaxation. Now let us round the fractional parts of $\mathbf{x}^*$ in the usual way, that is, values of 0.5 and up are rounded to 1 and smaller values down to 0. Let $\hat{\mathbf{x}}$ be the 0–1 solution obtained. First note that $\hat{\mathbf{x}}(v) \leq 2\mathbf{x}^*(v)$ for each $v \in V$. Also, for each $(u, v) \in E$, since $\mathbf{x}^*(u)+\mathbf{x}^*(v) \geq 1$, at least one of $\hat{\mathbf{x}}(u)$ and $\hat{\mathbf{x}}(v)$ must be set to 1. Hence $\hat{\mathbf{x}}$ is the incidence vector of a vertex cover of G whose total weight is within twice the total weight of the linear programming relaxation (which is a lower bound on the weight of the optimal vertex cover). Thus, we have a 2-approximate algorithm for this problem, which solves a linear programming relaxation and uses rounding to obtain a feasible solution.

The deterministic rounding of the fractional solution worked quite well for the vertex cover problem. One gets a lot more power from this approach by adding in randomization to the rounding step. Raghavan and Thompson [1987] proposed the following obvious randomized rounding scheme. Given a $0-1$ integer program, solve its linear programming relaxation to obtain an optimal $\mathbf{x}^*$. Treat the $\mathbf{x}_j{}^* \in [0, 1]$ as probabilities, i.e., let probability $\{\mathbf{x}_j = 1\} = \mathbf{x}_j{}^*$, to randomly round the fractional solution to a $0 - 1$ solution. Using Chernoff bounds on the tails of the binomial distribution, Raghavan and Thompson [1987] were able to show, for specific problems, that with high probability, this scheme produces integer solutions which are close to optimal. In certain problems, this rounding method may not always produce a feasible solution. In such cases, the expected values have to be computed as conditioned on feasible solutions produced by rounding. More complex (nonlinear) randomized rounding schemes have been recently studied and have been found to be extremely effective. We will see an example of nonlinear rounding in the context of semidefinite relaxations of the max-cut problem in the following.

15.7.2 Primal–Dual Approximation

The linear programming relaxation of the vertex cover problem, as we saw previously, is given by

$$(P_{VC})\quad \min\left\{\sum_{v\in V}\mathbf{w}(v)\mathbf{x}(v) : \mathbf{x}(u)+\mathbf{x}(v)\geq 1,\ \forall(u,v)\in E,\ \ \mathbf{x}(v)\geq 0\ \forall v\in V\right\}$$

and its dual is

$$(D_{VC})\quad \max\left\{\sum_{(u,v)\in E}\mathbf{y}(u,v) : \sum_{(u,v)\in E}\mathbf{y}(u,v)\leq w(v),\ \forall v\in V,\ \ \mathbf{y}(u,v)\geq 0\ \forall(u,v)\in E\right\}$$

The primal–dual approximation approach would first obtain an optimal solution $\mathbf{y}^*$ to the dual problem (D_{VC}). Let $\hat{V} \subseteq V$ denote the set of vertices for which the dual constraints are tight, i.e.,

$$\hat{V} = \left\{ v \in V : \sum_{(u,v) \in E} \mathbf{y}^*(u,v) = \mathbf{w}(v) \right\}$$

The approximate vertex cover is taken to be $\hat{V}$. It follows from complementary slackness that $\hat{V}$ is a vertex cover. Using the fact that each edge (u,v) is in the star of at most two vertices (u and v), it also follows that $\hat{V}$ is a 2-approximate solution to the minimum weight vertex cover problem.

In general, the primal–dual approximation strategy is to use a dual solution to the linear programming relaxation, along with complementary slackness conditions as a heuristic to generate an integer (primal) feasible solution, which for many problems turns out to be a good approximation of the optimal solution to the original integer program.

Remark 15.10 A recent survey of primal-dual approximation algorithms and some related interesting results are presented in Williamson [2000].

15.7.3 Semidefinite Relaxation and Rounding

The idea of using semidefinite programming to solve combinatorial optimization problems appears to have originated in the work of Lovász [1979] on the Shannon capacity of graphs. Grötschel et al. [1988] later used the same technique to compute a maximum stable set of vertices in perfect graphs via the ellipsoid method. Lovasz and Schrijver [1991] resurrected the technique to present a fascinating theory of semidefinite relaxations for general 0–1 integer linear programs. We will not present the full-blown theory here but instead will present a lovely application of this methodology to the problem of finding the maximum weight cut of a graph. This application of semidefinite relaxation for approximating MAXCUT is due to Goemans and Williamson [1994].

We begin with a quadratic Boolean formulation of MAXCUT

$$\max \left\{ \frac{1}{2} \sum_{(u,v) \in E} \mathbf{w}(u,v)(1 - \mathbf{x}(u)\mathbf{x}(v)) : \mathbf{x}(v) \in \{-1, 1\} \ \forall \ v \in V \right\}$$

where $G(V, E)$ is the graph and $\mathbf{w}(u,v)$ is the nonnegative weight on edge (u,v). Any $\{-1,1\}$ vector of $\mathbf{x}$ values provides a bipartition of the vertex set of G. The expression $(1 - \mathbf{x}(u)\mathbf{x}(v))$ evaluates to 0 if u and v are on the same side of the bipartition and to 2 otherwise. Thus, the optimization problem does indeed represent exactly the MAXCUT problem.

Next we reformulate the problem in the following way:

- We square the number of variables by substituting each $\mathbf{x}(v)$ with $\boldsymbol{\chi}(v)$ an n-vector of variables (where n is the number of vertices of the graph).
- The quadratic term $\mathbf{x}(u)\mathbf{x}(v)$ is replaced by $\boldsymbol{\chi}(u) \cdot \boldsymbol{\chi}(v)$, which is the inner product of the vectors.
- Instead of the $\{-1,1\}$ restriction on the $\mathbf{x}(v)$, we use the Euclidean normalization $\|\boldsymbol{\chi}(v)\| = 1$ on the $\boldsymbol{\chi}(v)$.

Thus, we now have a problem

$$\max \left\{ \frac{1}{2} \sum_{(u,v) \in E} \mathbf{w}(u,v)(1 - \boldsymbol{\chi}(u) \cdot \boldsymbol{\chi}(v)) : \|\boldsymbol{\chi}(v)\| = 1 \ \forall \ v \in V \right\}$$

which is a relaxation of the MAXCUT problem (note that if we force only the first component of the $\boldsymbol{\chi}(v)$ to have nonzero value, we would just have the old formulation as a special case).

The final step is in noting that this reformulation is nothing but a semidefinite program. To see this we introduce $n \times n$ Gram matrix Y of the unit vectors $\boldsymbol{\chi}(v)$. So $Y = X^T X$ where $X = (\boldsymbol{\chi}(v) : v \in V)$. Thus, the relaxation of MAXCUT can now be stated as a semidefinite program,

$$\max \left\{ \frac{1}{2} \sum_{(u,v)\in E} \mathbf{w}(u,v)(1 - Y_{(u,v)}) : Y \succeq 0, \quad Y_{(u,v)} = 1 \; \forall \; v \in V \right\}$$

Recall from Section 15.2 that we are able to solve such semidefinite programs to an additive error ϵ in time polynomial in the input length and $\log 1/\epsilon$ by using either the ellipsoid method or interior point methods.

Let $\boldsymbol{\chi}^*$ denote the near optimal solution to the semidefinite programming relaxation of MAXCUT (convince yourself that $\boldsymbol{\chi}^*$ can be reconstructed from an optimal Y^* solution). Now we encounter the final trick of Goemans and Williamson. The approximate maximum weight cut is extracted from $\boldsymbol{\chi}^*$ by randomized rounding. We simply pick a random hyperplane H passing through the origin. All of the $v \in V$ lying to one side of H get assigned to one side of the cut and the rest to the other. Goemans and Williamson observed the following inequality.

Lemma 15.1 *For $\boldsymbol{\chi}_1$ and $\boldsymbol{\chi}_2$, two random n-vectors of unit norm, let $\mathbf{x}(1)$ and $\mathbf{x}(2)$ be ± 1 values with opposing signs if H separates the two vectors and with same signs otherwise. Then $\tilde{E}(1 - \boldsymbol{\chi}_1 \cdot \boldsymbol{\chi}_2) \leq 1.1393 \cdot \tilde{E}(1 - \mathbf{x}(1)\mathbf{x}(2))$ where $\tilde{E}$ denotes the expected value.*

By linearity of expectation, the lemma implies that the expected value of the cut produced by the rounding is at least 0.878 times the expected value of the semidefinite program. Using standard conditional probability techniques for derandomizing, Goemans and Williamson show that a deterministic polynomial-time approximation algorithm with the same margin of approximation can be realized. Hence we have a cut with value at least 0.878 of the maximum cut value.

Remark 15.11 For semidefinite relaxations of mixed integer programs in which the integer variables are restricted to be 0 or 1, Iyengar and Cezik [2002] develop methods for generating Gomory–Chavatal and disjunctive cutting planes that extends the work of Balas et al. [1993]. Ye [2000] shows that strengthened semidefinite relaxations and mixed rounding methods achieve superior performance guarantee for some discrete optimization problems. A recent survey of semidefinite programming and applications is in Wolkowicz et al. [2000].

15.7.4 Neighborhood Search

A combinatorial optimization problem may be written succinctly as

$$\min\{f(x) : x \in X\}$$

The traditional neighborhood method starts at a feasible point x_0 (in X), and iteratively proceeds to a neighborhood point that is better in terms of the objective function f until a specified termination condition is attained. While the concept of neighborhood $N(x)$ of a point x is well defined in calculus, the specification of $N(x)$ is itself a matter of consideration in combinatorial optimization. For instance, for the traveling salesman problem the so-called *k-opt heuristic* (see Lin and Kernighan [1973]) is a neighborhood search method which for a given tour considers "neighborhood tours" in which k variables (edges) in the given tour are replaced by k other variables such that a tour is maintained. This search technique has proved to be effective though it is quite complicated to implement when k is larger than 3.

A neighborhood search method leads to a local optimum in terms of the neighborhood chosen. Of course, the chosen neighborhood may be large enough to ensure a global optimum but such a procedure is typically not practical in terms of searching the neighborhood for a better solution. Recently Orlin [2000]

has presented very large-scale neighborhood search algorithms in which the neighborhood is searched using network flow or dynamic programming methods. Another method advocated by Orlin [2000] is to define the neighborhood in such a manner that the search process becomes a polynomially solvable special case of a hard combinatorial problem.

To avoid getting trapped at a local optimum solution, different strategies such as tabu search (see, for instance, Glover and Laguna [1997]), simulated annealing (see, for instance, Aarts and Korst [1989]), genetic algorithms (see, for instance, Whitley [1993]), and neural networks have been developed. Essentially these methods allow for the possibility of sometimes moving to an inferior solution in terms of the objective function or even to an infeasible solution. While there is no guarantee of obtaining a global optimal solution, computational experience in solving several difficult combinatorial optimization problems has been very encouraging. However, a drawback of these methods is that performance guarantees are not typically available.

15.7.5 Lagrangian Relaxation

We end our discussion of approximation methods for combinatorial optimization with the description of Lagrangian relaxation. This approach has been widely used for about two decades now in many practical applications. Lagrangian relaxation, like linear programming relaxation, provides bounds on the combinatorial optimization problem being relaxed (i.e., lower bounds for minimization problems).

Lagrangian relaxation has been so successful because of a couple of distinctive features. As was noted earlier, in many hard combinatorial optimization problems, we usually have embedded some nice tractable subproblems which have efficient algorithms. Lagrangian relaxation gives us a framework to *jerry-rig* an approximation scheme that uses these efficient algorithms for the subproblems as subroutines. A second observation is that it has been empirically observed that well-chosen Lagrangian relaxation strategies usually provide very tight bounds on the optimal objective value of integer programs. This is often used to great advantage within partial enumeration schemes to get very effective pruning tests for the search trees.

Practitioners also have found considerable success with designing heuristics for combinatorial optimization by starting with solutions from Lagrangian relaxations and constructing good feasible solutions via so-called *dual ascent* strategies. This may be thought of as the analogue of rounding strategies for linear programming relaxations (but with no performance guarantees, other than empirical ones).

Consider a representation of our combinatorial optimization problem in the form

$$(P) \quad z = \min\{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b},\ \mathbf{x} \in X \subseteq \Re^n\}$$

Implicit in this representation is the assumption that the explicit constraints ($A\mathbf{x} \geq \mathbf{b}$) are *small* in number. For convenience, let us also assume that that X can be replaced by a finite list $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^T\}$.

The following definitions are with respect to (P):

- Lagrangian. $L(\mathbf{u}, \mathbf{x}) = \mathbf{u}(A\mathbf{x} - \mathbf{b}) + \mathbf{cx}$ where $\mathbf{u}$ are the Lagrange multipliers.
- Lagrangian-dual function. $\mathcal{L}(\mathbf{u}) = \min_{\mathbf{x}\in X}\{L(\mathbf{u}, \mathbf{x})\}$.
- Lagrangian-dual problem. $(D) \quad d = \max_{\mathbf{u}\geq 0}\{\mathcal{L}(\mathbf{u})\}$.

It is easily shown that (D) satisfies a weak duality relationship with respect to (P), i.e., $z \geq d$. The discreteness of X also implies that $\mathcal{L}(\mathbf{u})$ is a piecewise linear and concave function (see Shapiro [1979]). In practice, the constraints X are chosen such that the evaluation of the Lagrangian dual function $\mathcal{L}(\mathbf{u})$ is easily made (i.e., the *Lagrangian subproblem* $\min_{\mathbf{x}\in X}\{L(\mathbf{u}, \mathbf{x})\}$ is easily solved for a fixed value of $\mathbf{u}$).

Example 15.3

Traveling salesman problem (TSP). For an undirected graph G, with costs on each edge, the TSP is to find a minimum cost set H of edges of G such that it forms a Hamiltonian cycle of the graph. H is a Hamiltonian cycle of G if it is a simple cycle that spans all the vertices of G. Alternatively, H must satisfy:

(1) exactly two edges of H are adjacent to each node, and (2) H forms a connected, spanning subgraph of G.

Held and Karp [1970] used these observations to formulate a Lagrangian relaxation approach for TSP that relaxes the degree constraints (1). Notice that the resulting subproblems are minimum spanning tree problems which can be easily solved.

The most commonly used general method of finding the optimal multipliers in Lagrangian relaxation is subgradient optimization (cf. Held et al. [1974]). Subgradient optimization is the non differentiable counterpart of steepest descent methods. Given a dual vector $\mathbf{u}^k$, the iterative rule for creating a sequence of solutions is given by:

$$\mathbf{u}^{k+1} = \mathbf{u}^k + t_k \boldsymbol{\gamma}(\mathbf{u}^k)$$

where t_k is an appropriately chosen step size, and $\boldsymbol{\gamma}(\mathbf{u}^k)$ is a subgradient of the dual function $\mathcal{L}$ at $\mathbf{u}^k$. Such a subgradient is easily generated by

$$\boldsymbol{\gamma}(\mathbf{u}^k) = A\mathbf{x}^k - \mathbf{b}$$

where $\mathbf{x}^k$ is a maximizer of $\min_{\mathbf{x} \in X}\{L(\mathbf{u}^k, \mathbf{x})\}$.

Subgradient optimization has proven effective in practice for a variety of problems. It is possible to choose the step sizes $\{t_k\}$ to guarantee convergence to the optimal solution. Unfortunately, the method is not finite, in that the optimal solution is attained only in the limit. Further, it is not a pure descent method. In practice, the method is heuristically terminated and the best solution in the generated sequence is recorded. In the context of nondifferentiable optimization, the ellipsoid algorithm was devised by Shor [1970] to overcome precisely some of these difficulties with the subgradient method.

The ellipsoid algorithm may be viewed as a scaled subgradient method in much the same way as variable metric methods may be viewed as scaled steepest descent methods (cf. Akgül [1984]). And if we use the ellipsoid method to solve the Lagrangian dual problem, we obtain the following as a consequence of the polynomial-time equivalence of optimization and separation.

Theorem 15.8 *The Lagrangian dual problem is polynomial-time solvable if and only if the Lagrangian subproblem is. Consequently, the Lagrangian dual problem is $\mathcal{NP}$-hard if and only if the Lagrangian subproblem is.*

The theorem suggests that, in practice, if we set up the Lagrangian relaxation so that the subproblem is tractable, then the search for optimal Lagrangian multipliers is also tractable.

15.8 Prospects in Integer Programming

The current emphasis in software design for integer programming is in the development of shells (for example, CPLEX 6.5 [1999], MINTO (Savelsbergh et al. [1994]), and OSL [1991]) wherein a general purpose solver like branch and cut is the driving engine. Problem-specific codes for generation of cuts and facets can be easily interfaced with the engine. Recent computational results (Bixby et al. [2001]) suggests that it is now possible to solve relatively large size integer programming problems using general purpose codes. We believe that this trend will eventually lead to the creation of general purpose problem solving languages for combinatorial optimization akin to AMPL (Fourer et al. [1993]) for linear and nonlinear programming.

A promising line of research is the development of an empirical science of algorithms for combinatorial optimization (Hooker [1993]). Computational testing has always been an important aspect of research on the efficiency of algorithms for integer programming. However, the standards of test designs and empirical analysis have not been uniformly applied. We believe that there will be important strides in this aspect of integer programming and more generally of algorithms. J. N. Hooker argues that it may be useful to

stop looking at algorithmics as purely a deductive science and start looking for advances through repeated application of "hypothesize and test" paradigms, i.e., through empirical science. Hooker and Vinay [1995] developed a science of selection rules for the Davis–Putnam–Loveland scheme of theorem proving in propositional logic by applying the empirical approach.

The integration of logic-based methodologies and mathematical programming approaches is evidenced in the recent emergence of constraint logic programming (CLP) systems (Saraswat and Van Hentenryck [1995], Borning [1994]) and logico-mathematical programming (Jeroslow [1989], Chandru and Hooker [1991]). In CLP, we see a structure of Prolog-like programming language in which some of the predicates are constraint predicates whose truth values are determined by the solvability of constraints in a wide range of algebraic and combinatorial settings. The solution scheme is simply a clever orchestration of constraint solvers in these various domains and the role of conductor is played by resolution. The clean semantics of logic programming is preserved in CLP. A bonus is that the output language is symbolic and expressive. An orthogonal approach to CLP is to use constraint methods to solve inference problems in logic. Imbeddings of logics in mixed integer programming sets were proposed by Williams [1987] and Jeroslow [1989]. Efficient algorithms have been developed for inference algorithms in many types and fragments of logic, ranging from Boolean to predicate to belief logics (Chandru and Hooker [1999]).

A persistent theme in the integer programming approach to combinatorial optimization, as we have seen, is that the representation (formulation) of the problem deeply affects the efficacy of the solution methodology. A proper choice of formulation can therefore make the difference between a successful solution of an optimization problem and the more common perception that the problem is insoluble and one must be satisfied with the best that heuristics can provide. Formulation of integer programs has been treated more as an art form than a science by the mathematical programming community. (See Jeroslow [1989] for a refreshingly different perspective on representation theories for mixed integer programming.) We believe that progress in representation theory can have an important influence on the future of integer programming as a broad-based problem solving methodology in combinatorial optimization.

Defining Terms

Column generation: A scheme for solving linear programs with a huge number of columns.

Cutting plane: A valid inequality for an integer polyhedron that separates the polyhedron from a given point outside it.

Extreme point: A corner point of a polyhedron.

Fathoming: Pruning a search tree.

Integer polyhedron: A polyhedron, all of whose extreme points are integer valued.

Linear program: Optimization of a linear function subject to linear equality and inequality constraints.

Mixed integer linear program: A linear program with the added constraint that some of the decision variables are integer valued.

Packing and covering: Given a finite collection of subsets of a finite ground set, to find an optimal subcollection that is pairwise disjoint (packing) or whose union covers the ground set (covering).

Polyhedron: The set of solutions to a finite system of linear inequalities on real-valued variables. Equivalently, the intersection of a finite number of linear half-spaces in $\Re^n$.

ρ-Approximation: An approximation method that delivers a feasible solution with an objective value within a factor ρ of the optimal value of a combinatorial optimization problem.

Relaxation: An enlargement of the feasible region of an optimization problem. Typically, the relaxation is considerably easier to solve than the original optimization problem.

References

Aarts, E.H.L. and Korst, J.H. 1989. *Simulated annealing and Boltzmann machines: A stochastic approach to Combinatorial Optimization and neural computing*, Wiley, New York.

Ahuja, R. K., Magnati, T. L., and Orlin, J. B. 1993. *Network Flows: Theory, Algorithms and Applications*. Prentice–Hall, Englewood Cliffs, NJ.

Akgül, M. 1984. *Topics in Relaxation and Ellipsoidal Methods, Research Notes in Mathematics,* Pitman.

Alizadeh, F. 1995. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM J. Optimization* 5(1):13–51.

Applegate, D., Bixby, R. E., Chvátal, V., and Cook, W. 1994. Finding cuts in large TSP's. *Tech. Rep.* Aug.

Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M. 1992. Proof verification and hardness of approximation problems. In *Proc. 33rd IEEE Symp. Found. Comput. Sci.* pp. 14–23.

Balas, E., Ceria, S. and Cornuejols, G. 1993. A lift and project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming* 58: 295–324.

Barahona, F., Jünger, M., and Reinelt, G. 1989. Experiments in quadratic $0-1$ programming. *Math. Programming* 44:127–137.

Benders, J. F. 1962. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik* 4:238–252.

Berge, C. 1961. Farbung von Graphen deren samtliche bzw. deren ungerade Kreise starr sind (Zusammenfassung). Wissenschaftliche Zeitschrift, Martin Luther Universitat Halle-Wittenberg, Mathematisch-Naturwiseenschaftliche Reihe. pp. 114–115.

Berge, C. 1970. Sur certains hypergraphes generalisant les graphes bipartites. In *Combinatorial Theory and its Applications I.* P. Erdos, A. Renyi, and V. Sos eds., Colloq. Math. Soc. Janos Bolyai, 4, pp. 119–133. North Holland, Amsterdam.

Berge, C. 1972. Balanced matrices. *Math. Programming* 2:19–31.

Berge, C. and Las Vergnas, M. 1970. Sur un theoreme du type Konig pour hypergraphes. pp. 31–40. In *Int. Conf. Combinatorial Math.,* Ann. New York Acad. Sci. 175.

Bixby, R. E. 1994. Progress in linear programming. *ORSA J. Comput.* 6(1):15–22.

Bixby, R.E., Fenelon, M., Gu, Z., Rothberg, E., and Wunderling, R. 2001. M.I.P: Theory and practice: Closing the gap. Paper presented at Padberg-Festschrift, Berlin.

Bland, R., Goldfarb, D., and Todd, M. J. 1981. The ellipsoid method: a survey. *Operations Res.* 29:1039–1091.

Borning, A., ed. 1994. *Principles and Practice of Constraint Programming,* LNCS Vol. 874, Springer–Verlag.

Camion, P. 1965. Characterization of totally unimodular matrices. *Proc. Am. Math. Soc.* 16:1068–1073.

Cannon, T. L. and Hoffman, K. L. 1990. Large-scale zero-one linear programming on distributed workstations. *Ann. Operations Res.* 22:181–217.

Černikov, R. N. 1961. The solution of linear programming problems by elimination of unknowns. *Doklady Akademii Nauk* 139:1314–1317 (translation in 1961. *Soviet Mathematics Doklady* 2:1099–1103).

Chandru, V. and Hooker, J. N. 1991. Extended Horn sets in propositional logic. *JACM* 38:205–221.

Chandru, V. and Hooker, J. N. 1999. *Optimization Methods for Logical Inference,* Wiley Interscience.

Chopra, S., Gorres, E. R., and Rao, M. R. 1992. Solving Steiner tree problems by branch and cut. *ORSA J. Comput.* 3:149–156.

Chvátal, V. 1973. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Math.* 4:305–337.

Chvátal, V. 1975. On certain polytopes associated with graphs. *J. Combinatorial Theory B* 18:138–154.

Conforti, M. and Cornuejols, G. 1992a. Balanced 0, ± 1 matrices, bicoloring and total dual integrality. Preprint, Carnegie Mellon University.

Conforti, M. and Cornuejols, G. 1992b. A class of logical inference problems solvable by linear programming. *FOCS* 33:670–675.

Conforti, M., Cornuejols, G., and De Francesco, C. 1993. Perfect 0, ± 1 matrices. Preprint, Carnegie Mellon University.

Conforti, M., Cornuejols, G., Kapoor, A., and Vuskovic, K. 1994. Balanced 0, ± 1 matrices. Pts. I–II. Preprints, Carnegie Mellon University.

Conforti, M., Cornuejols, G., Kapoor, A. Vuskovic, K., and Rao, M. R. 1994. Balanced matrices. In *Mathematical Programming, State of the Art 1994.* J. R. Birge and K. G. Murty, Eds., University of Michigan.

Conforti, M., Cornuejols, G., and Rao, M. R. 1999. Decomposition of balanced 0, 1 matrices. *Journal of Combinatorial Theory* Series B 77:292–406.

Conforti, M. and Rao, M. R. 1993. Testing balancedness and perfection of linear matrices. *Math. Programming* 61:1–18.

Cook, W., Lovász, L., and Seymour, P., eds. 1995. *Combinatorial Optimization: Papers from the DIMACS Special Year*. Series in discrete mathematics and theoretical computer science, Vol. 20, AMS.

Cornuejols, G. and Novick, B. 1994. Ideal 0, 1 matrices. *J. Combinatorial Theory* 60:145–157.

CPLEX 6.5. 1999. Using the CPLEX Callable Library and CPLEX Mixed Integer Library, Ilog Inc.

Crowder, H., Johnson, E. L., and Padberg, M. W. 1983. Solving large scale 0–1 linear programming problems. *Operations Res.* 31:803–832.

Cunningham, W. H. 1984. Testing membership in matroid polyhedra. *J. Combinatorial Theory* 36B:161–188.

Dantzig, G. B. and Wolfe, P. 1961. The decomposition algorithm for linear programming. *Econometrica* 29:767–778.

Ecker, J. G. and Kupferschmid, M. 1983. An ellipsoid algorithm for nonlinear programming. *Math. Programming* 27.

Edmonds, J. 1965. Maximum matching and a polyhedron with 0–1 vertices. *J. Res. Nat. Bur. Stand.* 69B:125–130.

Edmonds, J. 1970. Submodular functions, matroids and certain polyhedra. In *Combinatorial Structures and their Applications,* R. Guy, Ed., pp. 69–87. Gordon Breach, New York.

Edmonds, J. 1971. Matroids and the greedy algorithm. *Math. Programming* 127–136.

Edmonds, J. 1979. Matroid intersection. *Ann. Discrete Math.* 4:39–49.

Edmonds, J. and Giles, R. 1977. A min-max relation for submodular functions on graphs. *Ann. Discrete Math.* 1:185–204.

Edmonds, J. and Johnson, E. L. 1970. Matching well solved class of integer linear polygons. In *Combinatorial Structure and Their Applications.* R. Guy, Ed., Gordon and Breach, New York.

Fonlupt, J. and Zemirline, A. 1981. A polynomial recognition algorithm for $K_4\backslash e$-free perfect graphs. *Res. Rep.,* University of Grenoble.

Fourer, R., Gay, D. M., and Kernighian, B. W. 1993. *AMPL: A Modeling Language for Mathematical Programming,* Scientific Press.

Fourier, L. B. J. 1827. In: Analyse des travaux de l'Academie Royale des Sciences, pendant l'annee 1824, Partie mathematique, *Histoire de l'Academie Royale des Sciences de l'Institut de France 7* (1827) xlvii-lv. (Partial English translation Kohler, D. A. 1973. *Translation of a Report by Fourier on his Work on Linear Inequalities. Opsearch* 10:38–42.)

Frank, A. 1981. A weighted matroid intersection theorem. *J. Algorithms* 2:328–336.

Fulkerson, D. R. 1970. The perfect graph conjecture and the pluperfect graph theorem. pp. 171–175. In *Proc. 2nd Chapel Hill Conf. Combinatorial Math. Appl.* R. C. Bose et al., Eds.

Fulkerson, D. R., Hoffman, A., and Oppenheim, R. 1974. On balanced matrices. *Math. Programming Study* 1:120–132.

Gass S. and Saaty, T. 1955. The computational algorithm for the parametric objective function. *Naval Research Logistics Quarterly* 2:39–45.

Gilmore, P. and Gomory, R. E. 1963. A linear programming approach to the cutting stock problem. Pt. I. *Operations Res.* 9:849–854; Pt. II. *Operations Res.* 11:863–887.

Glover, F. and Laguna, M. 1997. *Tabu Search*, Kluwer Academic Publishers.

Goemans, M. X. and Williamson, D. P. 1994. .878 approximation algorithms MAX CUT and MAX 2SAT. pp. 422–431. In *Proc. ACM STOC.*

Gomory, R. E. 1958. Outline of an algorithm for integer solutions to linear programs. *Bull. Am. Math. Soc.* 64:275–278.

Gomory, R. E. and Hu, T. C. 1961. Multi-terminal network flows. *SIAM J. Appl. Math.* 9:551–556.

Grötschel, M., Lovasz, L., and Schrijver, A. 1982. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica* 1:169–197.

Grötschel, M., Lovász, L., and Schrijver, A. 1988. *Geometric Algorithms and Combinatorial Optimization*. Springer–Verlag.

Haĉijan, L. G. 1979. A polynomial algorithm in linear programming. *Soviet Math. Dokl.* 20:191–194.
Held, M. and Karp, R. M. 1970. The travelling-salesman problem and minimum spanning trees. *Operations Res.* 18:1138–1162, Pt. II. 1971. *Math. Programming* 1:6–25.
Held, M., Wolfe, P., and Crowder, H. P. 1974. Validation of subgradient optimization. *Math. Programming* 6:62–88.
Hoffman, A. J. and Kruskal, J. K. 1956. Integral boundary points of convex polyhedra. In *Linear Inequalities and Related Systems,* H. W. Kuhn and A. W. Tucker, Eds., pp. 223–246. Princeton University Press, Princeton, NJ.
Hooker, J. N. 1988. Resolution vs cutting plane solution of inference problems: some computational experience. *Operations Res. Lett.* 7:1–7.
Hooker, J. N. 1992. Resolution and the integrality of satisfiability polytopes. Preprint, GSIA, Carnegie Mellon University.
Hooker, J. N. 1993. Towards and empirical science of algorithms. *Operations Res.* 42:201–212.
Hooker, J. N. and Vinay, V. 1995. Branching rules for satisfiability. In *Automated Reasoning* 15:359–383.
Huynh, T., Lassez C., and Lassez, J.-L. 1992. Practical issues on the projection of polyhedral sets. *Ann. Math. Artif. Intell.* 6:295–316.
IBM. 1991. *Optimization Subroutine Library—Guide and Reference (Release 2),* 3rd ed.
Iyengar, G. and Cezik, M. T. 2002. Cutting planes for mixed 0-1 semidefinite programs. *Proceedings of the VIII IPCO conference.*
Jeroslow, R. E. 1987. Representability in mixed integer programming, I: characterization results. *Discrete Appl. Math.* 17:223–243.
Jeroslow, R. E. and Lowe, J. K. 1984. Modeling with integer variables. *Math. Programming Stud.* 22:167–184.
Jeroslow, R. G. 1989. *Logic-Based Decision Support: Mixed Integer Model Formulation*. Ann. discrete mathematics, Vol. 40, North–Holland.
Jünger, M., Reinelt, G., and Thienel, S. 1995. Practical problem solving with cutting plane algorithms. In *Combinatorial Optimization: Papers from the DIMACS Special Year.* Series in discrete mathematics and theoritical computer science, Vol. 20, pp. 111–152. AMS.
Karmarkar, N. K. 1984. A new polynomial-time algorithm for linear programming. *Combinatorica* 4:373–395.
Karmarkar, N. K. 1990. An interior-point approach to NP-complete problems—Part I. In *Contemporary Mathematics,* Vol. 114, pp. 297–308.
Karp, R. M. and Papadimitriou, C. H. 1982. On linear characterizations of combinatorial optimization problems. *SIAM J. Comput.* 11:620–632.
Lawler, E. L. 1975. Matroid intersection algorithms. *Math. Programming* 9:31–56.
Lehman, A. 1979. On the width-length inequality, mimeographic notes (1965). *Math. Programming* 17:403–417.
Lin, S. and Kernighan, B.W. 1973. An effective heuristic algorithm for the travelling salesman problem. *Operations Research* 21: 498-516.
Lovasz, L. 1972. Normal hypergraphs and the perfect graph conjecture. *Discrete Math.* 2:253–267.
Lovasz, L. 1979. On the Shannon capacity of a graph. *IEEE Trans. Inf. Theory* 25:1–7.
Lovász, L. 1986. *An Algorithmic Theory of Numbers, Graphs and Convexity,* SIAM Press.
Lovasz, L. and Schrijver, A. 1991. Cones of matrices and setfunctions. *SIAM J. Optimization* 1:166–190.
Lustig, I. J., Marsten, R. E., and Shanno, D. F. 1994. Interior point methods for linear programming: computational state of the art. *ORSA J. Comput.* 6(1):1–14.
Martin, R. K. 1991. Using separation algorithms to generate mixed integer model reformulations. *Operations Res. Lett.* 10:119–128.
McDiarmid, C. J. H. 1975. Rado's theorem for polymatroids. *Proc. Cambridge Philos. Soc.* 78:263–281.
Megiddo, N. 1991. On finding primal- and dual-optimal bases. *ORSA J. Comput.* 3:63–65.
Mehrotra, S. 1992. On the implementation of a primal-dual interior point method. *SIAM J. Optimization* 2(4):575–601.

Nemhauser, G. L. and Wolsey, L. A. 1988. *Integer and Combinatorial Optimization*. Wiley.

Orlin, J. B. 2000. Very large-scale neighborhood search techniques. Featured Lecture at the International Symposium on Mathematical Programming, Atlanta, Georgia.

Padberg, M. W. 1973. On the facial structure of set packing polyhedra. *Math. Programming* 5:199–215.

Padberg, M. W. 1974. Perfect zero-one matrices. *Math. Programming* 6:180–196.

Padberg, M. W. 1993. Lehman's forbidden minor characterization of ideal 0, 1 matrices. *Discrete Math.* 111:409–420.

Padberg, M. W. 1995. *Linear Optimization and Extensions.* Springer–Verlag.

Padberg, M. W. and Rao, M. R. 1981. The Russian method for linear inequalities. Part III, bounded integer programming. Preprint, New York University, New York.

Padberg, M. W. and Rao, M. R. 1982. Odd minimum cut-sets and b-matching. *Math. Operations Res.* 7:67–80.

Padberg, M. W. and Rinaldi, G. 1991. A branch and cut algorithm for the resolution of large scale symmetric travelling salesman problems. *SIAM Rev.* 33:60–100.

Papadimitriou, C. H. and Yannakakis, M. 1991. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.* 43:425–440.

Parker, G. and Rardin, R. L. 1988. *Discrete Optimization.* Wiley.

Picard, J. C. and Ratliff, H. D. 1975. Minimum cuts and related problems. *Networks* 5:357–370.

Pulleyblank, W. R. 1989. Polyhedral combinatorics. In *Handbooks in Operations Research and Management Science.* Vol. 1, Optimization, G. L. Nemhauser, A. H. G. Rinooy Kan, and M. J. Todd, eds., pp. 371–446. North–Holland.

Raghavan, P. and Thompson, C. D. 1987. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica* 7:365–374.

Rhys, J. M. W. 1970. A selection problem of shared fixed costs and network flows. *Manage. Sci.* 17:200–207.

Saraswat, V. and Van Hentenryck, P., eds. 1995. *Principles and Practice of Constraint Programming,* MIT Press, Cambridge, MA.

Savelsbergh, M. W. P., Sigosmondi, G. S., and Nemhauser, G. L. 1994. MINTO, a mixed integer optimizer. *Operations Res. Lett.* 15:47–58.

Schrijver, A. 1986. *Theory of Linear and Integer Programming.* Wiley.

Seymour, P. 1980. Decompositions of regular matroids. *J. Combinatorial Theory* B 28:305–359.

Shapiro, J. F. 1979. A survey of lagrangian techniques for discrete optimization. *Ann. Discrete Math.* 5:113–138.

Shmoys, D. B. 1995. Computing near-optimal solutions to combinatorial optimization problems. In *Combinatorial Optimization: Papers from the DI'ACS special year.* Series in discrete mathematics and theoretical computer science, Vol. 20, pp. 355–398. AMS.

Shor, N. Z. 1970. Convergence rate of the gradient descent method with dilation of the space. *Cybernetics* 6.

Spielman, D. A. and Tang, S.-H. 2001. Smoothed analysis of algorithms: Why the simplex method usually takes polynomial time. *Proceedings of the The Thirty-Third Annual ACM Symposium on Theory of Computing*, 296–305.

Truemper, K. 1992. Alpha-balanced graphs and matrices and GF(3)-representability of matroids. *J. Combinatorial Theory* B 55:302–335.

Weyl, H. 1935. Elemetere Theorie der konvexen polyerer. *Comm. Math. Helv.* Vol. pp. 3–18 (English translation 1950. *Ann. Math. Stud.* 24, Princeton).

Whitley, D. 1993. *Foundations of Genetic Algorithms 2*, Morgan Kaufmann.

Williams, H. P. 1987. Linear and integer programming applied to the propositional calculus. *Int. J. Syst. Res. Inf. Sci.* 2:81–100.

Williamson, D. P. 2000. The primal-dual method for approximation algorithms. *Proceedings of the International Symposium on Mathematical Programming*, Atlanta, Georgia.

Wolkowicz, W., Saigal, R. and Vanderberghe, L. eds. 2000. *Handbook of semidefinite programming*. Kluwer Acad. Publ.
Yannakakis, M. 1988. Expressing combinatorial optimization problems by linear programs. pp. 223–228. In *Proc. ACM Symp. Theory Comput.*
Ye, Y. 2000. Semidefinite programming for discrete optimization: Approximation and Computation. *Proceedings of the International Symposium on Mathematical Programming*, Atlanta, Georgia.
Ziegler, M. 1995. *Convex Polytopes.* Springer–Verlag.